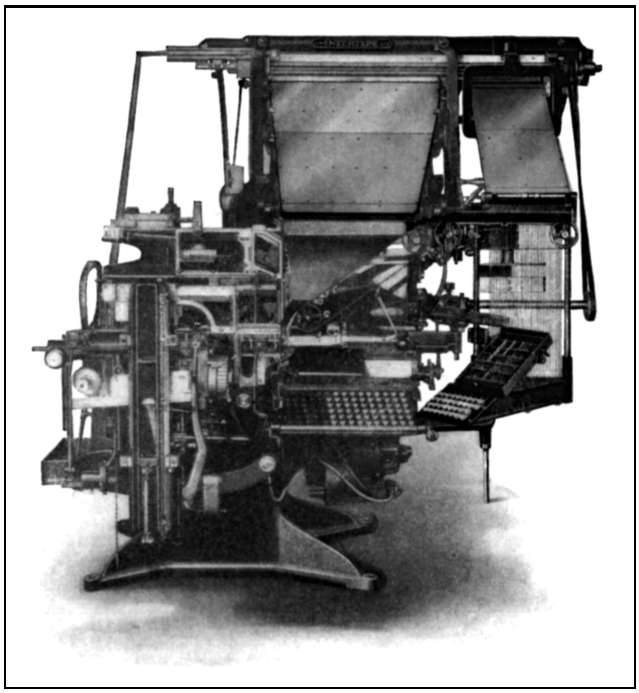
EuroTEX 2001

TEX AND META

THE GOOD, THE BAD AND THE UGLY



Proceedings

OF THE TWELFTH EUROPEAN TEX CONFERENCE

KERKRADE, THE NETHERLANDS

23–27 SEPTEMBER 2001

EuroTEX2001 is hosted by the Dutch language oriented TEX Users Group / Nederlandstalige TEX Gebruikersgroep. The webpage of the conference is at
`http://www.ntg.nl/eurotex/`

◊ ◊

Proceedings Editor
Simon Pepping

◊ ◊

Production
Taco Hoekwater ⋄ Siep Kroonenberg

◊ ◊

Design
Siep Kroonenberg

◊ ◊

Organizing Committee
Erik Frambach ⋄ Wybo Dekker ⋄ Luc De Coninck
Piet van Oostrum ⋄ Jules van Weerden

◊ ◊

Program Committee
Hans Hagen ⋄ Taco Hoekwater ⋄ Johannes Braams
Simon Pepping ⋄ Volker Schaa

◊ ◊

Sponsors
CAN-diensten ⋄ DANTE, Deutschsprachige Anwendervereinigung TEX e.V
Elvenkind BV ⋄ Focal Image Ltd ⋄ GUST, Polish TEX Users Group
GUTenberg, Groupe francophone des utilisateurs de TEX ⋄ H K Typesetting Ltd
Siep Kroonenberg ⋄ M&I/Stelvio BV ⋄ Mongolian TEX User Group
NTG, Dutch language oriented TEX users group ⋄ Pearson Education Uitgeverij BV
Sun Microsystems Nederland BV ⋄ TUG, TEX Users Group
UK-TuG, UK TEX Users' Group ⋄ Jules van Weerden

Table of Contents

Editorial, *Simon Pepping* 1

A note about the design of the Proceedings, *Siep Kroonenberg* . 5

Pattern generation revisited, *David Antoš and Petr Sojka* . . 7

Use of TEX Plugin technology for displaying of real-time weather and geographic information, *S. Austin, D. Menshikov and M. Vulis* 18

Texlib: a TEX reimplementation in library form, *Giuseppe Bilotta* 19

From database to presentation via Xml, Xslt and Context, *Berend de Boer* 27

Usage of MathML for paper and web publishing, *Tobias Burnus* 40

The Euromath system – a structured XML editor and browser, *J. Chlebíková, J. Guričan, M. Nagy and I. Odrobina* 41

Instant Preview and the TEX daemon, *Jonathan Fine* . . . 49

TEX and/or XML: good, bad and/or ugly, *Hans Hagen* . . . 59

TEX Top publishing, an overview, *Hans Hagen* 60

The bibliographic module for context, *Taco Hoekwater* . . . 61

mlBibTeX: a New Implementation of BibTeX, *Jean-Michel Hufflen* 74

Special Fonts, *Bogusław Jackowski and Krzysztof Leszczyński* . . 95

Metatype1: a MetaPost-based engine for generating Type 1 fonts, *Bogusław Jackowski, Janusz M. Nowacki, and Piotr Strzelczyk* 111

Natural TEX Notation in Mathematics, *Michal Marvan* . . . 120

TEX IN TEACHING, *Michael Moortgat, Richard Moot, Dick Oehrle* . . 130

POLIGRAF: FROM TEX TO PRINTING HOUSE, *Janusz Marian Nowacki* . 141

EXTENDING EXTEX, *Simon Pepping* 146

DIRECTIONS FOR THE TEXLIVE SYSTEM, *F. Popineau* 151

DCPIC, COMMUTATIVE DIAGRAMS IN A (LA)TEX DOCUMENT, *Pedro Quaresma de Almeida* 162

USING PDFTEX IN A PDF-BASED IMPOSITION TOOL, *Martin Schröder* . 173

ASCII-CYRILLIC AND ITS CONVERTER EMAIL-RU.TEX, *Laurent Siebenmann* 174

A TOUR AROUND THE NTS IMPLEMENTATION, *Karel Skoupy* . . . 187

VISUAL TEX: TEXLITE, *Igor Strokov* 188

CONVERSION OF TEX FONTS INTO TYPE1 FORMAT, *Péter Szabó* . . 192

MATH TYPESETTING IN TEX: THE GOOD, THE BAD, THE UGLY, *Ulrik Vieth* 207

'TYPOGRAPHY' AND PRODUCTION OF MANUSCRIPTS AND INCUNABULA, *Paul Wackers* 217

RE-INTRODUCING TYPE 3 FONTS TO THE WORLD OF TEX, *Włodek Bzyl* 219

LITERATE PROGRAMMING: NOT JUST ANOTHER PRETTY FACE, *M. A. Guravage* 244

♢ ♢ ♢

Editorial

Simon Pepping

We are pleased to present the Proceedings of the EUROTEX 2001 conference, the 12th annual gathering of the European TEX community. Like its predecessors, this conference has a varied and interesting programme, and thereby shows that its community is alive and thriving. Let us review what you will find in this volume.

No TEX conference would be complete without intense attention for fonts. Włodek Bzyl, Bogusław Jackowski, Janusz M. Nowacki and Piotr Strzelczyk make it clear that this is a strong tradition in the Polish TEX community. In two separate contributions they pay attention to the use of PostScript fonts with TEX. One of the contributions especially focuses on a new tool, METATYPE1, that should help bridge the gap between TEX's early font technology and the TYPE1 font technology that emerged later but has become the standard. Péter Szabó tries to do exactly the same with his new tool TEXtrace.

David Antoš and Petr Sojka revisit another TEX tool of the first hour: Pattern Generation. Since the birth of TEX the computing world has become truly international. They present a complete reimplementation that accommodates the needs of internationalization, and is written for the current situation in which extensibility and reusability have become much more important than managing a complex task in a tiny memory space.

Internationalization is also the goal of Jean-Michel Hufflen's reimplementation of another work horse of TEX, viz. BIBTEX.

Janusz M. Nowacki also presents a new version of his Poligraf package, which aids users in preparing their TEX work for printing in a professional printing house.

Bogusław Jackowski and Krzysztof Leszczyński revisit one of TEX's hooks for extensions: specials. Using a special pseudo-font, they try to make the insertion of specials more flexible.

Math typesetting is of course one of the main goals of TEX. Its intuitive syntax of entering formulae—for those who are familiar with them—has been one of the factors in its success. Nevertheless, there is always room for improvement. Michal Marvan will present his `nath` package for natural TEX notation in mathematics, which brings more intelligence into the interpretation of the typewritten formulae and their subsequent typesetting.

The same mathematicians who use formulae to express their ideas and work, often recognize that a diagram is more expressive. Several tools already exist to make drawing such diagrams easier. Pedro Quaresma explains how he has extended the

good and avoided the bad and ugly of earlier packages in his new package DCPic.

Ulrik Vieth applies the motto of this conference, the good, the bad, the ugly, to math typesetting in TEX. However good TEX is, it is always wise to keep an open and critical mind. Ulrik points out where TEX's math typesetting is good, but also where it has bad or even ugly elements. This critical appraisal will help us to better understand TEX's position in comparison with other software packages that aim to offer similar functionality, and especially when interoperability with such packages is an issue.

Literate programming is another gem that Knuth left the computing world. In this era, with its ever more complex software, good documentation is of paramount importance. Michael Guravage will show us if and how literate programming is still a useful technique for today's software developers.

To paraphrase Antoš and Sojka, TEX and friends, being nearly twenty years old, no longer completely suit today's needs. Therefore one of the recurring themes of this conference is reimplemenation. In this respect $\mathcal{NTS}$ has been with us for almost 10 years, first as an idea, and for the last three years as a work in progress. At the time of this conference it will be available in a β-release. Karel Skoupy, its developer, will take us on a tour through the program, and show us how $\mathcal{NTS}$ processes its input into its output. I myself will try to convince you that the release of $\mathcal{NTS}$ is an important mile stone: it is the first really existing TEX reimplemenation. We should use it to develop and experiment with extensions and new functionality.

Giuseppe Bilotta takes his own angle to reimplementation. He will discuss what is required when we want a new TEX that is also suited to provide immediate feedback to the user, WYSIWYG TEX.

When immediate feedback and WYSIWYG were provided by Word Processing software, it immediately appealed to the majority of users. In the TEX world this was often done away with a shrug, pointing to the inefficient way of working with these packages and to the high quality of TEX's output. But it cannot be denied that immediate feedback and WYSIWYG are desirable features, and some have started work to make this available within TEX as well. Igor Strokov has done extensive work on an implementation on MS Windows. Jonathan Fine has taken earlier work on IPC TEX further on the combination of Unix and Emacs. They both will show us what they have achieved so far.

A hot spot of development in the world around us is XML. Hans Hagen is building native support for XML documents into his macro package CONTEXT. He will tell us about this and other aspects of XML and TEX in one of his presentations.

Berend de Boer, early CONTEXT user, will demonstrate an application which uses CONTEXT's XML capabilities in combination with various other techniques: database, XML structuring, XSL styling, CONTEXT typesetting.

In the XML world, MATHML is the newly proposed standard for structuring mathematical formulae. Tobias Burnus will demonstrate how it can be used to publish work on paper and on the web.

While XML is a new effort of the computing world at large to work with structured input, in smaller circles this is not a new development at all, *vide* SGML or even LATEX. Work on editors that help the user to produce structured input also stands in

a tradition of a decade. In this tradition J. Chlebíková, J. Guričan, M. Nagy and I. Odrobina present their Euromath system, a structured XML editor and browser.

On earlier TEX conferences the extension arena was dominated by eTEX, later joined by pdfTEX and their combination pdfeTEX. At this conference they are notably absent. Hans Hagen has been one of the first TEX implementors to see the importance of these TEX extensions. Undoubtedly in his presentation on TEX TOP publishing these two will play their role as invaluable means by which Hans achieves his graphical finesse.

pdfTEX is here to stay, due to the interesting capabilities of the PDF format in conjunction with the PDF viewers. Martin Schröder and Tom Kacvinsky will both discuss the possibilities and the problems of the combination of PDF with TeX.

Another area where we have seen tremendous progress in the last years is that of TEX distributions. While in the early 90's it required considerable skill to set up a working TEX installation, the more recent distributions, among which the TEXLive distribution, have brought successful installation within the reach of an average user. Fabrice Popineau will discuss the problems that have to be overcome in putting together such a distribution. And he will show new directions for distributions, among which installation over a network.

Hans Hagen's CONTEXT macro package has attracted a large dedicated user base. As with all new applications, early users must do without utilities which are taken for granted with existing applications. Taco Hoekwater presents a bibliographic module for CONTEXT, which will provide CONTEXT users with a better integration of their favourite format and BIBTEX.

Besides the turmoil of the newest developments, we also take time to reflect on the developments in the past that have brought us to where we are now. Paul Wackers takes us back to the time when typography was still new and modelled itself after the existing industry of handwritten books. He shows us how the new technology slowly developed its own paradigms and style, much of which we still recognize.

Finally, why are we doing all this work? Surely, because it is interesting. But we also work to create a tool that users can deploy together with other tools to complete a complex task. Michael Moortgat, Richard Moot & Dick Oehrle do just that. Their focus is on a language technology project, and one of their requirements is high-quality, flexible typesetting of natural deductions. They will show how they use TEX successfully to meet that requirement.

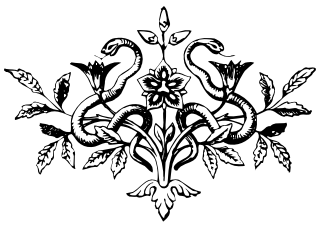
S. Austin, D. Menshikov, and M. Vulis demonstrate how they use TEX together with their own GeX plugin and PDF technology for displaying of real-time weather and geographic information.

Finally, Laurent Siebenmann goes back to the basics of TEX and applies that rare ability to write a program in TEX's macro language. With this technique he creates an application that allows users to read Russian emails, even when they do not have the required Cyrillic fonts installed.

Acknowledgement

Many articles in this Proceedings have benefited from critical review before publication. I thank Karel H. Wesseling, Michael Guravage, Taco Hoekwater and Johannes L. Braams for their efforts to review and correct submitted articles and to correspond with the authors about their suggestions for improvement.

Many contributors might not have decided to submit their contribution without the enthousiastic though urgent persuasion by Hans Hagen.



A note about the design of the Proceedings

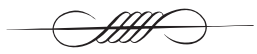
Siep Kroonenberg, siep@elvenkind.com

The theme of this conference being 'The Good, the Bad and the Ugly', I had a perfect excuse to loosen the reins of good taste and delve into all the ornamental fonts and artwork that is at my disposal. Taco and I settled for a nineteenth-century look: first, because of the exuberant graphic design from that period and second, because TEX typography and the Computer Modern font family already show a lot of affinity with nineteenth-century design.

We hoped that authors would pick up on this, and add their own flourishes to their contributions. There were some who did; see for yourself.

The titlepage illustration comes from a periodical 'Typografische Mededeelingen', Volume 15, issue 3, July 1919, published by Lettergieterij Amsterdam v/h N. Tetterode. It features a typesetting device called the Intertype. The descriptive text advertises its suitability for display type.

We like to thank Simon Pepping for the thorough job he did of preparing the electronic submissions; he made our work much, much easier than it might have been otherwise.



◇ ◇ ◇

*Pattern Generation Revisited**

David Antoš, Petr Sojka
Faculty of Informatics, Masaryk University Brno
Email: {xantos|sojka}@informatics.muni.cz

abstract.

The program PatGen, being nearly twenty years old, doesn't suit today's needs:

- ◇ it is nearly impossible to make changes, as the program is highly optimised (like TEX),
- ◇ it is limited to eight-bit encodings,
- ◇ it uses static data structures,
- ◇ reuse of the pattern technique and packed trie data structure for problems other than hyphenation (context dependent ligature handling, spell checking, Thai syllabification, etc) is cumbersome.

Those and other reasons explained further in the paper led us to the decision to reimplement PatGen from scratch in an object-oriented manner (like NTS–New Typesetting System reimplementation of TEX) and to create the PATtern LIBrary PatLib and the (hyphenation) pattern generator based on it.

We argue that this general approach allows the code to be used in many applications in computer typesetting area, in addition to those of pattern recognition, which include various natural language processing, optical character recognition, and others.

keywords: patterns, Unicode, hyphenation, tagging, transformation, Omega, PatGen, PatLib, reimplementation, templates, C++

Introduction

The ultimate goal of mathematics is to eliminate all need for intelligent thought.
— Graham, Knuth, Patashnik [2, page 56]

The ultimate goal of a typesetting engine is to automate as much as possible of what is needed for a given design, allowing the author to concentrate on the content of the text. The author maps her/his thoughts in *linear* writing, a sequence of *symbols*. Symbols (characters, words or even sentences) can be combined

*Presentation of the paper has been made possible with the support of EuroTEX bursary fund. This research has been partially supported by the Grant CEZ:J07/98:143300003.

into *patterns* (of characters, words or sentences). Patterns describe "higher rules" and dependencies between symbols, depending on *context*.

The technique of covering and inhibiting patterns used in the program PatGen [11] is highly effective and powerful. The pattern technique is an effective way to extract information out of large data files and to recognise the structures again. It is used in TEX as an elegant and language-independent solution for high-quality word hyphenation. This effective approach found its place in many other typesetting systems including the commercial ones. We think this method should be studied well, as many other applications are possible, in addition to those in the field of typesetting and natural language processing.

The generation of hyphenation patterns using the PatGen program does not satisfy today's needs. Many generalisations are needed for wider use. The Omega system [6, 12] was introduced. One of it's goals is to make direct typesetting of texts in Unicode possible, hence enabling the hyphenation of languages with more than 256 characters. An example of such a language is Greek, where 345 different combinations of Greek letters with accents, breathings, syllable lengths and the subscript iota are needed [5]. Therefore, Omega needs a generator capable of handling general/universal hyphenation patterns. Those new possibilities and needs in computer typesetting, together with the detailed analysis described below, led us to revise the usage of pattern recognition and to design new software to meet these goals.

The organisation of the paper is as follows. The next section (page 8) defines the patterns, using a standard example of hyphenation. Then an overview is given (page 9) of the process of pattern generation. The following section (page 10) describes one possible use for patterns and is followed by a section (page 11), in which the limitations for exploiting the current version of PatGen are argued.

The second part of this paper starts with a section (page 12) which describes the design of the new software library for pattern handling. Then packed digital trees, the basic data structure used in PatLib, are presented (page 12). Some thoughts about implementing the translation/tagging process using pattern based techniques are summarised in the section on page 15. The final section (page 16) contains a summary and suggestions for future work.

Patterns

> *Middle English* patron *'something serving as a model', from Old French. The change in sense is from the idea of a patron giving an example to be copied. Metathesis in the second syllable occured in the 16th century. By 1700* patron *ceased to be used on things, and the two forms became differentiated in sense.*
> — Origin of word *pattern*: [3]

Patterns are used to recognise "points of interest" in data. A point of interest may be the inter-character position where hyphenation is allowed, or the border between water and forest on a landscape photograph, or something similar. The

pattern is a sub-word of a given word set and the information of the points of interest is written between its symbols.

There are two possible values for this information. One value indicates the point of interest *is* here, the other indicates the point of interest *is not* here. Natural numbers are the typical representation of that knowledge; odd for yes, even for no. So we have *covering* and *inhibiting* patterns. Special symbols are often used, for example a dot for the word boundary.

Below we show a couple of hyphenation patterns, chosen out of the English hyphenating pattern file. For the purpose of the following example, we deal with a small subset of the real set of patterns. Note that the dot represents a word boundary.

```
.li4g    .lig5a    3ture    1ga    2gam
```

Using the patterns goes as follows. All patterns matching any sub-word of the word to be hyphenated are selected. Using the above subset of patterns with the word "ligature" we get:

```
. l i g a t u r e .
. l i4g
. l i g5a
        3t u r e
    1g a
```

The pattern `2gam` matches no sub-word of "ligature". The patterns *compete* and the endresult is the maximum for inter-character positions of all matching patterns, in our example we get:

```
. l0i4g5a3t0u0r0e .
```

According to the above we may hyphenate `lig-a-ture`.

To sum up: with a "clever" set of patterns, we are able to store a mapping from sequences of tokens (words) to an output domain — sequence of boolean values —, in our case positions of hyphenation points. To put it in another way: tokens (characters) emit output, depending on the context.

For a detailed introduction to TEX's hyphenation algorithms see [8, Appendix H]. We now need to know how patterns are generated to understand why things are done this way.

Pattern Generation

> *An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil's behaviour.*
> — Alan Turing, [16]

Generating a *minimal* set of competing patterns completely covering a given phenomenon is known to be NP-complete. Giving up the minimality requirement, we may get surprisingly good results compressing the *input data* information into a pattern set iteratively. Let us now describe the generating process.

We need a large input data file with marked points of interest. Hyphenating words, we use a large dictionary with allowed hyphenation points. Now we repeat going through the data file in several *levels*. We generate covering patterns in odd levels and inhibiting ones in even levels.

We have a rule how to choose *pattern candidates* at each level. In our case it may be "an at most k characters long substring of the word containing the hyphenation point". We choose pattern candidates and store them into a suitable data structure. Not all candidates are good patterns, so we need a *pattern choosing rule*. Usually we remember the number of times when the candidate helps and spoils finding a correct hyphenation point. We always test new candidates according to all patterns selected so-far. We are interested in the functionality of the whole set. The pattern choosing rule may be a linear function over the number of good/bad word efficiency of the candidate compared to a threshold. This heuristic is used in PatGen, but other heuristics may lead to better (e.g. with respect to space) pattern sets with the same functionality. The candidates marked as good by the previous process are included into the set of patterns. The pattern set still makes mistakes. We continue generating another level, an even level this time, when we create inhibiting patterns. The next level will be covering and so on. A candidate at a certain level is good if it repairs errors made by previous levels.

This is also the way how PatGen works. A PatGen user has no chance to change the candidate and/or pattern choosing rules, which are similar to the ones previously described. Hyphenating patterns for TEX have been created for several dozens of languages [15], usually created from a list with already hyphenated words. There are languages where the patterns were created by hand, either entirely, or in part.

How successful is this technique? The natural language dictionary has several megabytes of data. Out of such a dictionary patterns of tens of kilobytes may be prepared, covering more than 98 % of the hyphenation points with an error rate of less than 0.1 %. Experiments show that four or five levels are enough to reach those parameters. Using various strategies of setting linear threshold parameters we may optimise the patterns to size, covering ratio and/or errors [13]. As not many lists with hyphenated words are publicly available for serious research on pattern generation heuristics, we think that most available patterns are suboptimal. For more information on pattern generation using PatGen have a look at tutorial [4].

Tagging with Patterns

The solution of the hyphenation problem and the techniques involved have been studied extensively [15] and together with long-lasting usage in TEX and other typesetting systems, their advantages have been verified. The application of the techniques of bootstrapping and stratification [13, 14] made them even more attractive. However, to the best of our knowledge, sofar nobody has suggested and used a context dependent task for the resolution of other ambiguities.

We may look at the hyphenation problem as a problem of *tagging* the possible hyphenation positions in finite sequences of characters called words. On a different level of abstraction, the recognition of sentence borders is nothing more than "tagging" the begins and ends of sentences in sequences of words.

Yet another example: in quality typography, it is often necessary to decide, whether a given sequence of characters is to be typeset as a ligature (such as ij, fi, fl) and not as separate characters (ij, fi, fl). This ambiguity has to be resolved by the tagging of appropriate occurences, depending on the context: ligatures are e.g. forbidden on compound word boundaries.

All these tasks (and many others, see page 15) are "isomorphic" — the same techniques developed and used for hyphenation may be used here as well. The key issue in applicability of the techniques for the variety of context-dependent tagging tasks is the understanding and effective implementation of the pattern generation process. The current implementation of PatGen is not up to these possible new uses.

PatGen Limitations

What man wants is simply independent choice,
whatever that independence may cost
and wherever it may lead.
— Fedor Dostoevsky, *Notes from Underground* (1864)

The program PatGen has several serious restrictions. It is a monolithic structured code, which, although very well documented (documented Pascal, WEB), is very difficult to change. PatGen is also "too optimised", necessary to make it possible to run in the core of the PDP-10, so understanding the code is not easy. In this sense PatGen is very similar to TEX itself. The data structures are tightly bound to the stored information: high-level operations are performed on the data structures directly without any levels of abstraction.

The data structures of PatGen are hardwired for eight-bit characters. Modification to handle more characters — full Unicode — is not straightforward. The maximum number of PatGen levels is nine. When generating patterns, you can collect candidates of the same length at the same time only. The data structures are static, running out of memory requires the user to change constants in the source code and recompile the program.

Of course PatGen may be used to generate patterns on other phenomenons besides word hyphenation, but only if you transform the problem into hyphenation. This might be non-trivial and moreover, it's feasible only for problems with small alphabets, less than approximately 240 symbols (PatGen uses some ascii characters for special and output purposes).

PatLib

My library was dukedom large enough.
— Shakespeare, The Tempest (1611), act 1, sc. 2 l. 109

We decided to generalise PatGen and to implement the PATtern LIBrary PatLib for general pattern manipulation. We hope that this will make the techniques easily accessible. A Unicode word hyphenation pattern generator is the testbed application.

For portability and efficiency reasons we chose C^{++} as the implementation language. The C^{++} code is embedded in CWEB to keep the code documented as much as possible. Moreover the code "patterns" called *templates* in C^{++} let us postpone the precise type specification to higher levels of development which turned out to be a big advantage during the step-wise analysis. We do hope that templates increase flexibility of the library.

The PatLib library consists of two levels, the finite language store (which is a finite automaton with output restricted to finite languages, implemented using packed trie) and the pattern manipulator. The language store handles only basic operations over words, such as inserting, deleting, getting iteratively the whole stored language and similar low-level operations. The output of a word is an object in general, so is the input alphabet.

The pattern manipulator handles patterns, it means words with multiple positioned outputs. We also prepared a mechanism to handle numbers of good and bad counts for pattern candidates.

The manipulator and the language store work with objects in general, nevertheless to keep efficiency reasonable we suggest to use numbers as internal representation for the external alphabet. Even if the external alphabet is Unicode, not all Unicode characters are really used in one input data file. So we can collect the set of all used characters and build a bijection between the alphabet and the internal representation by numbers $\{1, \ldots, n\}$, where all the numbers are really used.

We separated the semantics from the representation. We don't have to care what the application symbols are. An application using this library may implement any strategy for the generation of patterns.

Of course we have to pay for more generality and flexibility with performance loss. As an example, the output of a pattern in PatGen is a pointer to a hash table containing pairs $\langle \text{level_number}, \text{position} \rangle$, we must have an object with a copy constructor. At the time of writing of this article we are unable to give an indication of the performance ratio.

Packed digital tree (trie)

Gentle reader, if you are not interested in programming or data structures, feel free to skip this section. It will do no harm for understanding the rest of the article. The trie data structure we use to store patterns is quite known.

Its practically usable variant — being described only seldom in programming books — is much less known.

A trie is usually presented and described as in [9]: it is an m-ary tree, its nodes are m-ary vectors indexed by a sorted finite alphabet. A node in depth l from the root corresponds to the prefix of length l. Finding a word in a trie starts at the root node. We take the next symbol of the word, let it be k. Then the kth member of the node points to the lower level node, which corresponds to the unread rest of the word. If the word is not in the trie, we get the longest prefix.

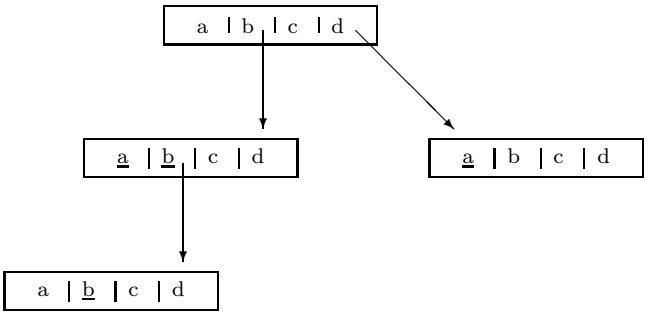


FIGURE 1: TRIE — AN EXAMPLE

Figure 1 shows a trie containing the words *ba*, *bb*, *bbb*, and *da* over the alphabet $\{a, b, c, d\}$. Underlining indicates the end of a word.

It is not difficult to implement this data structure. Nodes may be put into a linear array one by one, pointers going to the start of the next nodes. But this approach wastes memory, especially if the words are long and the nodes sparse. Using dynamic memory does not change this property.

The advantage of a trie is that the time needed for the look-up and inserting of a word is *linear to the length of the word*, this means the time needed does not depend on the amount of words stored.

The need for memory may be reduced (paying with a small amount of time), as shown by Liang in [10]. In practical applications the nodes are sparse, hence we want to store them *mixed into one another* into a linear array. One node uses the fields which are left empty by another node.

When working with this structure, we must have a way to decide which field belongs to a certain node. This may be done with a little trick. To each field we add information about which alphabet symbol is related to the array position. Moreover two nodes must never start at the same position of the array. We must add one bit of information if the position is a *base position* and when inserting, we never pack two nodes at the same base position.

Index	1	2	3	4	5	6	7	8	9
Character		a	b	c	d	a	b	b	a
Pointer			5		8		6		
Base position?	Y				Y	Y		Y	
End of word?						Y	Y	Y	Y

FIGURE 2: PACKED TRIE

In Figure 2 the same language as used previously is stored. The trie starts on position 1, this is the base position. The trie root is treated specially for implementation reasons, it is always stored fully in the array, even if there are no words starting with the appropriate character. Only the pointer is null in that case.

We assert numerical values to the alphabet symbols: $a = 1$, $b = 2$, $c = 3$, $d = 4$. How do we distinguish the fields belonging to a node? Let the node start at base position z. We go through positions $z + a, \ldots, z + d$ and check where the condition "the character on position $z+i$ is i" holds. For the root this is always true. In the root, there is a pointer under character b (on position 3). It points to the base position 5. Moreover the root says we have a word starting with d. Let us go through the positions belonging to base position 5, this means related to the prefix b. They are:

- ◇ position 6, this should be related to a, this holds, the pointer is null, the end-of-word flag is true, hence ba belongs to the language and any other word starting with ba does not.
- ◇ position 7, which is related to b, so the position belongs to the node, the position is end-of-word, therefore bb belongs to the language and there are words starting with bb continuing as said by the node on base position 6.
- ◇ positions 8 and 9 should belong to the characters c and d, this is not the case, these positions do not belong to the current node.

The reader may easily check that the table contains the same language as shown in Figure 1. Sixteen fields are needed to store the language naïvely, we need nine when packing. The ratio is not representative, it depends on language stored.

The trie nodes may be packed using the first-fit algorithm. This means when packing a node, we find the first position where it can be done, where we do not interfere with existing nodes and we do not use the same base position. We can speed up the process using the following heuristics. If the node we want to store is filled less than a threshold, we don't loose time finding an appropriate position but store it at the end of the array. Otherwise we use the first-fit method as described. Our experience shows that array usage much better than 95 % may be obtained without significant loss of speed.

Pattern Translation Processes

If all you have is a hammer, everything looks like a nail.
— popular aphorism

Let us review several tasks related to computer typesetting, in order to see whether they could be implemented as a Pattern Translation Processes (PTP), implemented using PatLib. Most of them are currently being tackled via *external* ΩTPs in Omega [7].

Hyphenation of compound words The plausibility of the approach has been shown for German in [13].

Context-dependent ligatures In addition to the already mentioned ligatures at the compound word boundaries, another example exists:

Fraktur long s versus short s In the Gothic letter-type there are two types of s-es, a long one and the normal one. The actual usage depends on the word morphology. Another typical context-dependent auto-tagging procedure implementable by PTP.

End of sentence recognition To typeset a different width space at the end of a sentence automatically, one has to filter out abbreviations that do not normally appear at the end of a sentence. A hard, but doable task for PTP.

Spell checking Storing a big word-list in a packed digital tree is feasible and gives results comparable to spelling checkers like ispell. For languages with inflection, however, several hierarchical PTP's are needed for better performance. We are afraid that PTP's cannot beat specialised fine-tuned morphological analysers, though.

Thai segmentation There is no explicit word/sentence boundary, punctuation and inflexion in Thai text. This information, implicitly tagged by spaces and punctuation marks in most languages, is missing in standard Thai text transliteration. It is, however, needed, during typesetting for line-breaking. It has yet to be shown that pattern-based technology is at least comparable to the currently used probabilistic trigram model [1].

Arabic letter hamza Typesetting systems for Arabic scripts need to have built-in logic for choosing one of five possible appearances of the letter hamza, depending on context. This process can easily be formulated as a PTP.

Greek accents In [7, page 153] there is an algorithm — full of exceptions and context dependent actions — for the process of adding proper accents in Greek texts. Most parts of it can easily be described as a sequence of pattern-triggered actions and thus be implemented as a PTP.

Similarly, there are many Czech texts written without diacritics from the times when email mailers only supported seven-bit ASCII , which wait to be converted into proper form. Even for this task PTP's could be trained.

We believe that PTP implementation based on PatLib could become common ground for most, if not all, ΩTP's. Hooking and piping various PTP's in Omega may lead to uniform, highly effective (all those mapping are *linear* with respect to the length of the text) document processing. Compared to external ΩTP's, PTP imple-

mentation would win in speed. To some extent, we think that a new version of PatGen based on PatLib will not only be independent of language (for hyphenation), but of application, too.

Summary and Future Work

Write once, use everywhere.
— paraphrase of a well known slogan

We have discussed the motivation for developing a new library for the handling and generation of patterns, and we presented its design and first version. We argue that the pattern-based techniques have a rich future in many application areas and hope for PatLib to be playing a rôle there.

Readers are invited to download the latest version of PatLib and the PatGen reimplementation at `http://www.fi.muni.cz/~xantos/patlib/`.

Acknowledgement
The authors thank a reviewer for detailed language revision.

References

[1] Orchid corpus. Technical Report TR-NECTEC-1997-001, Thai National Electronics and Computer Technology Center, 1999. `http://www.links.nectec.or.th/`.

[2] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, USA, 1989.

[3] Patrick Hanks, editor. *The New Oxford Dictionary of English*. Oxford University Press, Oxford, 1998.

[4] Yannis Haralambous. A Small Tutorial on the Multilingual Features of PATGEN2. in electronic form, available from CTAN as `info/patgen2.tutorial`, January 1994.

[5] Yannis Haralambous and John Plaice. First applications of Ω: Adobe Poetica, Arabic, Greek, Khmer. *TUGboat*, 15(3):344–352, September 1994.

[6] Yannis Haralambous and John Plaice. Methods for Processing Languages with Omega. In *Proceedings of the Second International Symposium on Multilingual Information Processing, Tsukuba, Japan*, 1997. available as `http://genepi.louis-jean.com/omega/tsukuba-methods97.pdf`.

[7] Yannis Haralambous and John Plaice. Traitement automatique des langues et composition sous Omega. *Cahiers Gutenberg*, (39–40):139–166, May 2001.

[8] Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[9] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1998.

[10] Franklin M. Liang. *Word Hy-phen-a-tion by Com-put-er*. Ph.D. Thesis, Department of Computer Science, Stanford University, August 1983.

[11] Franklin M. Liang and Peter Breitenlohner. `PAT`tern `GEN`eration program for the TEX82 hyphenator. Electronic documentation of PATGEN program version 2.3 from web2c distribution on CTAN, 1999.

[12] John Plaice and Yannis Haralambous. The latest developments in OMEGA. *TUGboat*, 17(2):181–183, June 1996.

[13] Petr Sojka. Notes on Compound Word Hyphenation in TEX. *TUGboat*, 16(3):290–297, 1995.

[14] Petr Sojka. Hyphenation on Demand. *TUGboat*, 20(3):241–247, 1999.

[15] Petr Sojka and Pavel Ševeček. Hyphenation in TEX — Quo Vadis? *TUGboat*, 16(3):280–289, 1995.

[16] Alan Turing. Computing machinery and intelligence. *Mind*, (59):433–460, 1950.

♢ ♢ ♢

Use of TEX plugin technology for displaying of real-time weather and geographic information

S. Austin[a], D. Menshikov[b], and M. Vulis[a]
[a] Dept. of CSC, CCNY, NY, USA
[b] MicroPress, Inc, USA

In this article we show how by means of the GeX plugin technology one can process and display geographic information including real-time weather data as part of a TEX→PDF compilation.

The plugin technology [introduced at TUG2000] functions under the PDF backend of the VTeX compiler; it allows the user to enhance the TEX-integrated PostScript converter (GeX) with user-defined language extensions. Plugins can be used for plotting or retrieving specialized data; earlier plugin examples were used for business or scientific plots.

The Tiger plugin is a new experimental plugin which can retrieve static geographic data (the Tiger database, for instance), as well as the real time weather data and plot them together within the context of a TeX document compilation; it can be used, for example, to supplement static TeX documents (papers, books) with maps, as well as (on a server environment) produce real-time weather maps.

♢ ♢ ♢

TEXlib: a TEX reimplementation in library form

Giuseppe Bilotta

Introduction

I first came across the need for a TEX in library form when I was thinking about developing a graphical real-time front-end to TEX (the TEXPerfect project, more info at `http://texperfect.sourceforge.net`). A quick survey (on the `comp.text.tex` newsgroup) showed that other projects could have benefited from a library providing TEX typesetting capabilities, and I thus decided to develop TEXlib as a separate project from TEXPerfect. A "call for developers" on the same newsgroup provided the project with developers/consultants/helpers.

An analysis of the current opinion on TEX and its future added another aim to the TEXlib project: since we're reimplementing TEX, why shouldn't we take the occasion to break through TEX's limitations?

The current status of TEX's "future" is the following. We have some extensions:

- ε-TEX, mainly concerned with TEX's parser (eyes and mouth), but also providing right-to-left typesetting;
- Omega, mainly concerned with TEX's typesetting routines (guts), but also with an innovative input parser (OCPlists);
- pdfTEX, mainly concerned with TEX's output routines, but also prodiving new typesetting features (one for all: hanging characters);

and some new implementations:

- *NTS*, in Java, close to its first version;
- ANT, in Scheme, still in a very early stage (but still more complete than TEXlib`;-)`.

More than once it has been suggested that the three TEX extensions should blend into one; well, TEXlib might as well be the point of convergence. The reasons behind such an expectation are mainly

- the library form of TEXlib: while a library can be easily used with a command line interface, it is much harder to let a command line driven program act as a library;
- being the youngest and less-formed TEX-based project, TEXlib can deal with all the issues of integration of extensions, without requiring changes to (not-yet-existant) sources. This assumes that the ε-TEX, pdfTEX and Omega developers are willing to provide feedback and suggestions on how to integrate the various features provided by the different extensions.

Library structure

The library will be split in three parts: input parser modules (eyes & mouth), typesetting module (guts), output modules. The three parts will be kept as separate as possible (thus allowing things like an XML input parser with an Omega typesetting engine and producing PDF output).

The typesetting module interface will be public, so that custom input parser and output producers could take advante of TEX's typesetting capabilities. The typesetting module will basically provide two functions: the paragraph builder (accepting a horizontal list and returning a vertical list) and the page builder (accepting a vertical list and returning two vertical lists).

Vertical and horizontal lists will be built by the input parsing modules, and can be sent to the output modules to produce "real" output (DVI pages, PDF documents, some other format specifically tuned for real-time preview, etc). The library will provide default input parsers and output producers.

Assuming the library-provided modules will be used, the following is more or less how a typical session of TEXlib would run.

1. TEXlib is loaded (if not loaded already).
2. A TEXlib "context"[1] is initialized, providing a format file, and various settings, such as: which extensions are allowed, what kind of input is provided (TEX, XML), what kind of output is expected (PDF, DVI, memory output), how many pages to "cache" in memory, etc.
3. The main function will be provided with the address of the buffer containing the data.
4. Variables for the context are initialized.
5. Typesetting Loop.
6. Feedback Loop.

Library loading and library instantiation are kept separate, not only to allow the library to be shared among clients, but also to allow the same client–library link to make use of different TEXlib contexts (useful if typesetting a mixed TEX/XML document, for example).

TEX input parsing approach

While the current TEX extensions deal with *what* TEX does, the main concern of TEXlib is *how* TEX is supposed to do it.

Currently, TEX works in a sequential way: source code is input one line at a time, and the data is sequentially processed to ship out some kind of output (DVI file, one page at a time, plus various auxiliary files, depending on the format used).

Much of this processing mode is somehow required by TEX's *embedded macro* (programming) capabilities (or, conversely, TEX's macro capabilities were built with this workflow in mind). This means that no *revert* is possible: once a change (`\def`, catcode change, etc) has been made, the only way to "roll back" is through another

[1] "context" is the internal name of a library instantiation.

"forward" change, restoring the previous value (which must have been saved somewhere, usually in another macro).

Since the main application of TEXlib is TEX real-time editing, and since editing (and especially reviewing) happens in a non-sequential way, we need a way to allow moving backward and forward through the source.

Mainly, we can consider two approaches:

1. ONE-STACK-PER-THING-TO-BE-SAVED APPROACH.
   The idea behind this approach is to push the old value each time a change is made, and then pop it when rolling back.
2. CHECK-POINT STACK APPROACH.
   This approach can be thought of as "intermediate dumping": fix a check-point (say, at page ship out), and push/pop *all* the values (also the unchanged ones) each time the user crosses the check-point.

Functionality. Pros and cons

Let's consider approach 1 first. The technique works as follows. A token is examined and 'executed'. If the execution (complete macro expansion or execution of a primitive) changes some values, the library stores, at the end of the execution, the original values together with the new ones. This allows easy back-tracking, and restart of compilation from an arbitrary point.

The largest problems are memory usage (but a comparison between this method and the next one would need some real-world cases) and the complexity connected with `\let` and `\def` when applied to or otherwise influencing the same token that causes the change. Currently, the best idea I can think of is to simply wait until macro expansion and argument scanning ends, before saving the values, but I still couldn't think of a robust way to implement such an idea.

Let's now have a look at approach 2. Probably the best place to insert a checkpoint is at page shipout. The library then acts this way: it parses data until it fills the page cache; then it waits for client feedback; if data needs to be re-parsed, the library re-enters the typesetting loop.

1. Typesetting Loop.
   a. (Parsing and typesetting) If no data available goto 1.e else read next token and execute it.
   b. If shipout goto 1.c else goto 1.a.
   c. (Shipout) Save $checkpoint = (line, col), page, memory_dump$. Increase $cached_pages$.
   d. If $cached_pages = max_cache$ goto 1.e else goto 1.a.
   e. Tell client that we finished our job.
2. Feedback Loop.
   a. The client tells the library where the cursor is.
   b. If cursor crosses a *checkpoint*, reload $memory_dump$ for the entered page.
   c. If a change has been made (a token has been inserted), goto 2.d else goto 2.a
   d. Set $line, col$ to that of the latest *checkpoint* and goto 1.

This approach is relatively easy to implement, but quite memory consuming (consider e.g. that a typical ConTEXt format file is between 4 and 5 megabytes in size, and this

amount of memory should be allocated for each cached page). Also, it is quite slow when rolling back before the first cached page, since in this case typesetting would have to start right from the beginning of the file (when the first cached page is around page 100, this would mean that we need to retypeset $100 - max_cache$ pages). These two problems could be minimized with

1. dynamic memory allocation;
2. "unbalanced" cached pages.

The idea behind 1 is that memory should only be allocated when needed, thus giving smaller memory hits (and higher performance) for typical jobs, while still allowing heavy jobs to be done without reinitializing the library.

The idea behind feature 2 is to keep more "back" pages than "forward" pages in cache. For example, if there are 10 cached pages, the current page is likely to be the 8th or 9th cached page (provided that we are past page 8 in the document). If the user is scrolling backwards, the library will restart compiling before the user hits the 1st cached page (say, when the user gets to the 5th cached page), discarding "forward" pages (say, from the 7th to the 10th) and it will stop compilation three pages before the first cached page.

The various settings (number of cached pages (10), number of back (7) and forward (2) pages and the discarding treshold (5)) should be user configurable, possibly at runtime. A future version might have auto-detection of "best" suggested settings.

Another shortcoming in this approach, at least when fixing checkpoints at page shipout, is TEX's asynchronous page shipout. When a page is actually shipped out, TEX can already be quite a few source lines past the last source line on the page being shipped out. A possible solution could be to save two source coordinates instead of one: the $(line, col)$ pair of the data that caused the shipout, and the $(line, col)$ pair of the last data contained in the shipout.

Of course it is to be seen if there is some way to determine the last data shipped out, and the parent $(line, col)$ coordinate.

This problem is tightly connected with the synchronization of source and view. Consider that it was TEXPerfect that pushed me into developing TEXlib. Since TEXPerfect will likely run in split-view, with the output in the upper half and the source in the lower half, we need a way to synchronize cursor positions in the view with cursor positions in the source, and the synchronization has to be as precise as possible.

Synchronization

Synchronization needs a continuous feedback between client and library. On one side we have the client, which provides the source, the current position within the source, the modification status. On the other side, the library provides the output (in the specified form) and its status. But there is another important kind of feedback that the library can provide to the client (and we will see shortly why it is important): tokenization of the input lines. This means that for each input line read, the library should return where each token starts, where it ends and which subsequent tokens are being "eaten up".

Why is this important? Let's consider the first level of synchronization: source specials. At least for the current page (but possibly for each cached page) the library should know the originating $(line, col)$ coordinate for each output bit (character, rule, glue).

This can be memory-optimized by taking advantage of one-to-one correspondance: for example, in the case of a paragraph containing only characters, it is only important to know where the first letter originated from.

But there are cases of multiple tokens providing one or no bits of input (think of multiple spaces, or some kind of assignments), and conversely of single tokens providing more than one bit of input. To make things more complicated, most command tokens are multi-letter, and they can take arguments.

The idea is then to inform the client about this. Thus, tokens scanned during macro expansion will be given a particular status, so that the client knows which tokens will go "directly" to the output, and which should be considered as arguments of macros. The client can then take appropriate actions: for example, commands (both the command token and its arguments) could appear as a button in the source code window, and be skipped with a single keystroke while browsing (instead of requiring a keystroke for each character composing the token and its arguments).

Problems

There are some intrinsic problems that are inherent to a librarization of TEX, and they can be summarized as follows:

A. Error management

This answers the question: how to handle input-parsing errors?

Documents fed to the library could be split in two categories: 'hand-written' documents and 'machine-written' documents.

A hand-written document is simply a document written with a standard editor. A common source of error in such a case could be of the kind "`\hobx` instead of `\hbox`" (that is, all the kinds of error that arise from typos during source-writing).

Machine-written documents, on the other hand, are documents where commands are inserted in the source by the editor *only*, just like it happens, for example, with word-processors: the user selects *Italic* in the font properies (or presses an appropriate shortcut) and the client inserts the appropriate codes into the source.

Such a document will be free of typo-like errors (unless the editor has been badly programmed). But still, other kinds of error are possible (for example, fragile commands in moving arguments).

Since we are implementing a library, when such an error occurs the library would inform the client of the fact; acording to the spirit of batch processing in TEX, a suggested solution will be proposed. It is then up to the client to

choose what to do: consult the user, provide its own solution or simply enact what has been proposed by the library.

A useful option that the client *should* provide is "verbatim reparsing" from where the error occurred. For example, if the error was an caused by an undefined csname, the action would be to consider the csname to be a sequence of character tokens.

B. Input/Output management.
There are also other input/output issues. Two different approaches should be taken, distinguishing user I/O from file I/O.

Management of user input/output will be entirely left on the client side: the library will inform the client when user input is requested, and simply defer logging information to the client. The client is then free to report the logging information and the query to the users, or simply hide them. For example, in the case of a query to the user the client might ask the user for an answer, and then provide that answer as default each time the same query is met again during re-typesetting.

There are input/output issues connected with external file management, too. During re-typesetting of source files `\input`s and `\write`s referring to the same data will be met again and again. How is this to be managed?

First of all it is important to differentiate between user-provided `\input` (for example, when dealing with master documents and subdocuments) and "system" `\input` (for example, input of auxiliary files, as requested by recent formats like LaTeX and ConTeXt).

Since the client is the only one who knows if an `\input` is user- or system-provided, it should be left to the client to decide which `\input`s and `\write`s should be honoured and which not. The library will have to manage the input/output in a rational way, with complete knowledge on which data was output by which command, so as to be able to remove that data before insertion of the new data.

(This issue is still blurry and to be discussed. See also issue D.)

A separate problem is finally provided by the `\write18` primitive. But this will probably come up later (for example, if `\write18` is used to call Metapost, a possible solution is to just pass the request to the Metapost library, when it will be implemented).

C. Extensibily.
Should TeXlib be extensible? To what extent, how easily? This is a serious problem: I don't want to stimulate proliferation of many incompatible TeXlib extensions, while I still believe that time will show its limitations and thus the need to overcome those. This will mean that there will be an "official" TeXlib, with "official" extensions. (In other words, more ConTeXt-like development than a LaTeX-like development).

D. "Backward" compatibility and auxiliary files.
TEX format files use auxiliary files to pass information between a compilation and the next. Such auxiliary files are created during typesetting, sometimes post-processed, and finally re-input on subsequent compilations. Most of the time the data stored by auxiliary files is data provided later in the document but physically used earlier.

Such a way to pass information back and forth is required in sequential data parsing, and cannot be easily overcome when subverting the sequential paradigm while still keeping source compatibility.

There are though some things that can be done to solve some issues.

One issue is, for example, the (possible) need to change the stored data each time the writing command is issued. This might lead to physical abuse of storage supports (disks), and can be circumvented with intelligent data analysis (storing the data if and only if) and file caching (keeping the auxiliary file in memory instead of on disk —this requires the library to know which files are auxiliary ones).

Another issue is the actual usage of the data stored in the auxiliary file. This data is usually input at the beginning of the document, and this gives some problems.

First of all, it is useless to re-input the same data each time the user crosses an input command: a more intelligent way to deal with this is to check if the data has changed and act consequently.

The second important issue is the following: assume that the auxiliary file is actually input when needed (for example, when the cursor enters the table of contents), the same input may create differences in subsequent pages, thus producing differences in the auxiliary file, and possibly (in case of non-converging changes) cause a library lock-up.

I propose the following solution (inspired by the way a famous word-processor works). During normal editing/previewing auxiliary files are not dealt with, and input/output requests to it are simply ignored. At the user's request, though, a series of *sequential* compilations take place (the document is *generated*). The information collected during these sequential compilations is then stored in an auxiliary file and used.

Future ideas

A. Convergence of TEX extensions.
As mentioned in the introduction, the "planning" status of TEXlib encourages it as a point of convergence for TEX extensions. Cooperation with the developers of the other TEX extensions will render this actual.

B. Librarization of TEX's friends (*at least* Metapost).
The TEXlib project is not involved with TEX alone, but with the whole family. Librarization of the other members of the family —or of their successors— will

allow a previously unseen integration of the components —with all the power that comes from it.

C. Possible integration of TEX and Metapost in library form.
Library Metapost is a top priority (after TEX itself) of TEXlib; Among other reasons, because of the impressive power derived from the intregration of TEX and Metapost (it is even possible to emulate a poor-man's Omega, at least when referring to multidirectional typesetting capabilities).

D. Further extensions of TEX.
TEX is finally showing its age. While still outperforming other similar or related programs (from word processors to desktop publishing programs) in many aspects, there are features which simply cannot be implemented robustly without providing new core features. Some (maybe most) of these features can be implemented through the use of specials and appropriate output postprocessors, but native implementation of them could make the whole thing robust, standard and fast.

Some proposed extensions are the following:

- ⋆ *Native color support (and other attibutes).*
  This has been discussed (and will be discussed again when the time comes) on the TEXlib developers mailing list (and no conclusion has been reached). We came to the conclusion that it could be nice to "load" each node type with attributes not directly related to typesetting. Color is an example of such an attribute, since a character is put in the same place whatever its color is.
- ⋆ *Multiple reference points.*
  This has been requested by Hans Hagen, to ease and make more robust the management of `\vtop` and similar boxes.
- ⋆ *Code tables management.*
  Most of the internal code tables (catcodes, lccodes, uccodes, sfcodes etc.) of TEX (and of some of its extensions) have to be manually changed value-by-value each time such a change is needed. Internal support for saving/restoring (partial) code tables would speed up things like font- and language-switching. This feature probably needs to be made cooperative with Omega's ocplists.

♢ ♢ ♢

From database to presentation via XML, XSLT and ConTEXt

Berend de Boer

Introduction

Much data exists only in databases. A familiar example is an address list. Every once in a while this data must be presented to humans. To continue with the address list example, annually an address list must be printed and mailed.

In this article I attempt to given an exhaustive overview of going from structured data through ConTEXt to output, see figure 1.

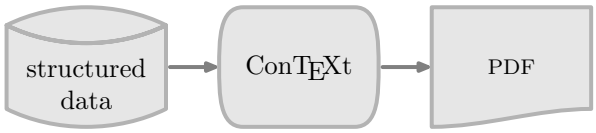


Figure 1 Going from data through ConTEXt to output

As any data format can be represented by XML, this document focuses on typesetting data in XML in ConTEXt, see figure 2. When the data is in XML, in can be directly handled by ConTEXt. ConTEXt has a built-in XML typesetting engine that can handle XML tags just fine. You don't have to convert the XML to ConTEXt macro's first. This is the subject of the following section.

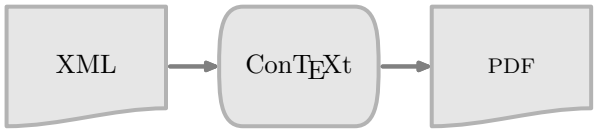


Figure 2 Going from xml through ConTEXt to output

When the data is not yet in XML format, is has to be converted to XML. 'Converting comma ...' covers converting comma separated data to XML. 'Converting relational

. . .' covers converting data residing in relational databases such as DB/2 and InterBase to XML. 'Typesetting sql . . .' covers going from such data straight to ConTEXt without converting to XML first.

The XML data you have might not be easy to typeset. An advantage of XML is that it is easy to transform into XML with a different format. There is a specific language, XSLT, to transform XML into XML, see figure 3. This is the subject of 'Transforming XML ...'.

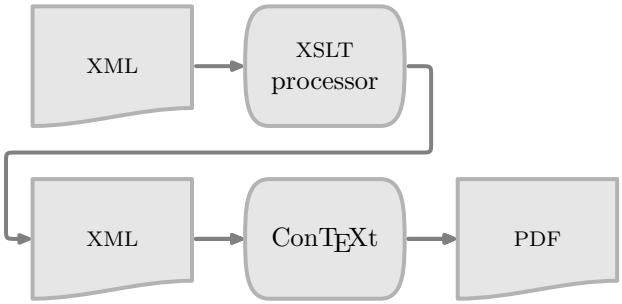


Figure 3 Going from xml through ConTEXt to output

Typesetting xml in ConTEXt

This section assumes that the data to be typeset is already available in XML. The next sections cover converting data to XML.

For this article a special XML format was chosen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE rows SYSTEM "example.dtd">
<rows>
  <row>
    <field>Re-introduction of Type 3 fonts into the TeX world</field>
    <field>Wlodzimierz Bzyl</field>
  </row>
  <row>
    <field>The Euromath System - a structure XML editor and browser</field>
    <field>J. Chlebkov, J. Gurican, M. Nagy, I. Odrobina</field>
  </row>
  <row>
    <field>Instant Preview and the TeX daemon</field>
    <field>Jonathan Fine</field>
  </row>
</rows>
```

This example files shows the first three entries in the abstract list of euroTEX 2001 at the time of this writing. The DTD of this XML file is:

```
<!-- DTD used for examples in article "From database to presentation
     via XML, XSLT and ConTeXt". -->

<!ELEMENT rows (row*)>

<!ELEMENT row (field*)>

<!ELEMENT field (#PCDATA)>
```

I still prefer DTDs above XML Schema's. They're far more readable and you can't express all well–formed XML files with XML Schema's anyway, so what's the advantage?

Our examples have the root tag `<rows>`. Our examples can have 0 or more `<row>` tags. Each `<row>` tag can have zero or more `<field>` tags.

With ConTEXt we can typeset this with the `\processXMLfilegrouped` macro:

```
\processXMLfilegrouped {example.xml}
```

The result of this is:

Re-introduction of Type 3 fonts into the TeX world Wlodzimierz Bzyl The Euromath System - a structure XML editor and browser J. Chlebkov, J. Gurican, M. Nagy, I. Odrobina Instant Preview and the TeX daemon Jonathan Fine

As you can see, this gives us just the plain text, no formatting is done. We can typeset our XML in a table with adding the following definitions and processing it again:

```
\defineXMLenvironment [rows]    \bTABLE \eTABLE
\defineXMLpickup       [row]     \bTR \eTR
\defineXMLpickup       [field]  \bTD \eTD

\processXMLfilegrouped {example.xml}
```

These definitions bind the start and end of a tag to a certain ConTEXt macro. Our result is now:

Re-introduction of Type 3 fonts into the TeX world	Wlodzimierz Bzyl
The Euromath System - a structure XML editor and browser	J. Chlebkov, J. Gurican, M. Nagy, I. Odrobina
Instant Preview and the TeX daemon	Jonathan Fine

The above example uses the new table environment of ConTEXt. As this specific environment cannot yet split across pages, the tabulate environment is a better choice for typesetting data. For this environment we need the following definitions:

```
\defineXMLpickup [rows]   {\starttabulate[|p(6cm)|p|]} \stoptabulate
\defineXMLpickup [row]    \NC \NR
\defineXMLpickup [field]  \relax \NC

\processXMLfilegrouped {example.xml}
```

Our result is now:

Re-introduction of Type 3 fonts into the TeX world	Wlodzimierz Bzyl
The Euromath System - a structure XML editor and browser	J. Chlebkov, J. Gurican, M. Nagy, I. Odrobina
Instant Preview and the TeX daemon	Jonathan Fine

I hope I've made clear the basic ideas of typesetting XML:

1. Make sure the XML data is in a proper tabular format.
2. Define mappings to the ConTEXt table, tabular or TABLE environment.
3. Use `\processXMLfilegrouped` to process your XML file.

Converting comma separated files to xml

Not always is data in the proper format. This section and the next cover converting non XML data into XML data.

Many programs can give CSV (Comma Separated Variable) data as output. An example of this format is:

```
"Fred","Flintstone",40
"Wilma","Flintstone",36
"Barney","Rubble",38
"Betty","Rubble",34
"Homer","Simpson",45
"Marge","Simpson",39
"Bart","Simpson",11
"Lisa","Simpson",9
```

In this format, fields are separated by comma's. String fields can be surrounded by double quotes. In XML this data should look like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE rows SYSTEM "example.dtd">
<rows>
<row>
 <field>Fred</field>
 <field>Flintstone</field>
 <field>40</field>
</row>
<row>
 <field>Wilma</field>
```

```
 <field>Flintstone</field>
 <field>36</field>
</row>
<row>
 <field>Barney</field>
 <field>Rubble</field>
 <field>38</field>
</row>
<row>
 <field>Betty</field>
 <field>Rubble</field>
 <field>34</field>
</row>
<row>
 <field>Homer</field>
 <field>Simpson</field>
 <field>45</field>
</row>
<row>
 <field>Marge</field>
 <field>Simpson</field>
 <field>39</field>
</row>
<row>
 <field>Bart</field>
 <field>Simpson</field>
 <field>11</field>
</row>
<row>
 <field>Lisa</field>
 <field>Simpson</field>
 <field>9</field>
</row>
</rows>
```

Converting CSV to our 'standard' XML format can be done by a simple Perl script:

```
#!/usr/bin/perl -w use strict;

# test arguments
if (@ARGV == 0)
{
    die "Supply a filename as argument";
}

use Text::ParseWords;

open INPUT, "$ARGV[0]" or die "Can't open input file $ARGV[0]: $!";

print "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n";
print "<!DOCTYPE rows SYSTEM \"example.dtd\">\n";
```

```
print "<rows>\n";
while (<INPUT>) {
    chop;
    my @fields = quotewords(",", 0, $_);
    print "<row>\n";
    my $i = 0;
    foreach $field (@fields) {
        print "\t<field>$field</field>\n";
        $i++;
    }
    print "</row>\n";
}
print "</rows>\n";
```

Use this script as follows:

```
perl -w csv2xml.pl flintstones.csv > flintstones.xml
```

If you don't know what Perl is, you can read more about it at `http://www.perl.org`. Most UNIX users have Perl installed by default. Windows or Macintosh users can download Perl at `http://www.cpan.org/ports/index.html`. I'm not a particular fan of Perl, I can't remember the syntax if I've not used it for a few days. However, you can count on it being available for almost all operating systems.

Converting relational (SQL) data to XML

Much of this worlds data resides in relational databases. It is not difficult to retrieve data from a relational database and turn it into XML.

Consider the following SQL table:

```
create table "family member" (
  "id_family member" smallint not null primary key,
  "surname" character varying(30) not null,
  "family name" character varying(40) not null,
  "age" smallint not null);
```

And the following insert statements:

```
insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (1, 'Fred', 'Flintstone', 40);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (2, 'Wilma', 'Flintstone', 36);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (3, 'Barney', 'Rubble', 38);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (4, 'Betty', 'Rubble', 34);
```

```
insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (5, 'Homer', 'Simpson', 45);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (6, 'Marge', 'Simpson', 39);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (7, 'Bart', 'Simpson', 11);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (8, 'Lisa', 'Simpson', 9);
```

A simple ANSI SQL query to extract the data and sort it in surname is:

```
select surname, age
  from flintstone
  order by surname
```

SQL output is usually not returned in XML format, and certainly not in the format we've described in the previous section. Here is the output that is generated by InterBase:

```
Database:  flintstones.gdb

surname                         age
============================== =======

Barney                           38
Bart                             11
Betty                            34
Fred                             40
Homer                            45
Lisa                              9
Marge                            39
Wilma                            36
```

Before embarking on our tour to make this SQL more ConTEXt friendly, let's first explore how to get such output. Most relational databases offer a command line tool which can execute a given query. Frequently this tool is called `isql`. To present the above example I called `isql` as follows:

```
opt/interbase/bin/isql flintstones.gdb -i select1.sql -o select1.out
```

The actual InterBasequery, instead of the ANSI query presented above, looked like:

```
select "surname", "age"
  from "flintstone"
  order by "surname";
```

At the end of this section I present the command line tools of PostgreSQL and DB2.

There are two methods to typeset SQL output in ConTEXt:

1. Embed XML tags in the `select` statement.
2. Embed ConTEXt macro's in the select statement.

The first approach will be discussed in this section, the latter approach in the next section.

Embedding XML in a select statement to generate the format discussed before can be done with this InterBase `select` statement:

```
select
    '<row><field>',
    "surname",
    '</field><field>',
    "age",
    '</field></row>'
  from "flintstone"
  order by
    "surname";
```

The first two rows of the output look like this (slightly formatted for clarity):

```
Database:  flintstones.gdb

                surname                   age
====== ======= ======== ======== ======= ===== ======== ======

<row>  <field> Barney   </field> <field>    38 </field> </row>
<row>  <field> Bart     </field> <field>    11 </field> </row>
```

There are five problems with the output of InterBase `isql`, four of which are present in the above output:

1. There is no container tag, i..e the `<rows>` tag is missing.
2. The first line contains the database used: `flintstones.gdb`.
3. Column headers are present.
4. InterBase inserts columns headers after every 20 lines. Because there are just a few flintstones, this does not show up in my example, but I've typesetted thousands of entries, and there you have to deal with it. Fortunately, this can be easily solved by using the `-page` parameter and calling `isql` as follows:

   ```
   isql flintstones.gdb -i select1.sql -o select1.out -page 32000
   ```

   This will insert a column headers only every 32000 rows.
5. There is a lot of superfluous white space. White space is usually not a problem with TEX, and it also isn't with ConTEXt's XML typesetting macro's. I consider this a feature. If white space is a problem, you can attempt to write a somewhat different SQL statement like:

   ```
   select
     '<row><field>' + surname + '</field><field>' + age + '</field></row>'
     from flintstones
   ```

   This example uses string concatenation instead of putting the XML tags in different columns.

The first three problems cannot be solved by some parameter. We have to use Perl again. Here my script to remove the column headers of an InterBase SQL output file and at the appropriate container tag:

```
#!/usr/bin/perl -w use strict;

# test arguments
if (@ARGV == 0)
{
    die "Supply a filename as argument";
}

open INPUT, "$ARGV[0]" or die "Can't open input file $ARGV[0]: $!";

# read up to the line with ====
while (<INPUT>) {
    if (/^=.*/) {
        last;
    }
};

# skip one more line
<INPUT>;

# now just dump all input to output
print "<rows>\n";
while (<INPUT>) {
    print;
}
print "</rows>\n";
```

The output is now a lot more like XML:

```
<rows>
<row><field> Barney                </field><field>      38 </field></row>
<row><field> Bart                  </field><field>      11 </field></row>
<row><field> Betty                 </field><field>      34 </field></row>
<row><field> Fred                  </field><field>      40 </field></row>
<row><field> Homer                 </field><field>      45 </field></row>
<row><field> Lisa                  </field><field>       9 </field></row>
<row><field> Marge                 </field><field>      39 </field></row>
<row><field> Wilma                 </field><field>      36 </field></row>

</rows>
```

We can typeset this with:

```
\defineXMLpickup [rows]
  {\starttabulate[|p(7cm)|p|] \HL\NC surname \NC age \NC\NR\HL}
  {\stoptabulate}
\defineXMLpickup [row]
  \NC \NR
\defineXMLpickup [field]
  \relax \NC
```

```
\processXMLfilegrouped {select2.xml}
```

And the result looks great!

surname	age
Barney	38
Bart	11
Betty	34
Fred	40
Homer	45
Lisa	9
Marge	39
Wilma	36

As promised here the commands to extract data from DB2 and PostgreSQL. For DB2 use the `db2` command, like this:

```
db2 -td\; -f myfile.sql -r myfile.out
```

The `-td` option defines the command separator character. I use the ';' character for this. After the `-f` option follows an SQL file with one or more `select` statements. With the `-r` option you can redirect the output to a file.

PostgreSQL has the `psql` to extract SQL data. Use it like this:

```
psql -d flintstones -f myfile.sql -o myfile.out
```

The `-d` option specified the database name. The `-f` option specifies the file with the `select` statements. The `-o` option redirects the output to a file.

Typesetting sql without generating xml

In the previous section SQL output was enhanced with XML tags. The XML tags were then mapped to ConTEXt macro's. It is possible to skip the XML tag generation by directly putting the ConTEXt commands in the SQL `select` statement:

```
select
    '\NC',
    "surname",
    '\NC',
    "age",
    '\NC\NR'
  from "flintstone"
  order by
    "surname";
```

From the output we again have to remove the lines we don't need. This can be done with more or less a Perl script like the one shown before. It can be even simpler as it doesn't have to add something before or after the data. After cleaning up the output should look like:

```
\NC    Barney                    \NC        38 \NC\NR
\NC    Bart                      \NC        11 \NC\NR
\NC    Betty                     \NC        34 \NC\NR
\NC    Fred                      \NC        40 \NC\NR
\NC    Homer                     \NC        45 \NC\NR
\NC    Lisa                      \NC         9 \NC\NR
\NC    Marge                     \NC        39 \NC\NR
\NC    Wilma                     \NC        36 \NC\NR
```

The ConTEXt code to typeset the data in this case is:

```
\starttabulate[|p(7cm)|p|]
\HL
\NC surname \NC age \NC\NR
\HL
\input select3.tex
\stoptabulate
```

Transforming xml with xslt

In the preceding section we've seen how XML can be generated from non XML sources. This section is concerned with generating XML that can be typeset in ConTEXt from existing XML sources. Usually XML sources are not in a format that can be typeset easily. Such XML has to be transformed to the XML format presented earlier. Fortunately there is an entire language devoted to transforming XML to XML. It is called XSLT, a quite complete and not too difficult language. More information about XSLT can be found at `http://www.w3.org/Style/XSL/`.

The first example is making a list of euroTEX 2001 authors and their presentations. The program listing in XML at time of this writing looked like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<program>
  <day weekday="Monday" date="24 September 2001">
    <item time="9.00h"><opening/></item>
    <item time="9.15h">
      <presentation>
        <author>Hans Hagen</author>
        <title>Overview of presentations</title>
      </presentation>
    </item>
    <item time="9.45h">
      <presentation>
        <author>Wlodzimierz Bzyl</author>
        <title>Re-introduction of Type 3 fonts into the TeX world</title>
      </presentation>
    </item>
    <break time="10.30h" type="coffee"/>
    <item time="11.00u">
      <presentation>
```

```
        <author>Michael Guravage</author>
        <title>Literate Programming: Not Just Another Pretty Face</title>
      </presentation>
    </item>
    </day>
</program>
```

With the following XSL stylesheet we can transform this to our standard XML format:

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/program">
  <rows><xsl:text>&#xa;</xsl:text>
  <xsl:apply-templates select="day/item/presentation"/>
  </rows><xsl:text>&#xa;</xsl:text>
</xsl:template>

<xsl:template match="presentation">
  <row>
    <field><xsl:value-of select="author"/></field>
    <field><xsl:value-of select="title"/></field>
  </row><xsl:text>&#xa;</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

This transformation gives us something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<rows>
<row><field>Hans Hagen</field><field>Overview of presentations</field></row>
<row><field>Karel Skoupy</field><field>NTS implementation</field></row>
</rows>
```

How we can typeset this, should be clear enough by now! It is probably more helpful to explain the XSL stylesheet a bit. An XSL stylesheet usually consists of many `<xsl:template>` tags. The XSL processor takes the first one that matches the root node (the '/' separator) as the main template. It starts the transformation there (The real rules are somewhat more difficult, but not important here). In our case we match the `</program>` node. We output the `<rows>` tag and next we output all the presentations. This is done with a `<xsl:apply-templates>` tag that searches for a template that matches the selected nodes. In the template that matches the presentation node, we output the `<row>` tag and the individual fields.

An XSL processor can do many advanced things with XML, see figure 4. It cannot only generate XML, but also straight ConTEXt code for example, or just plain text.

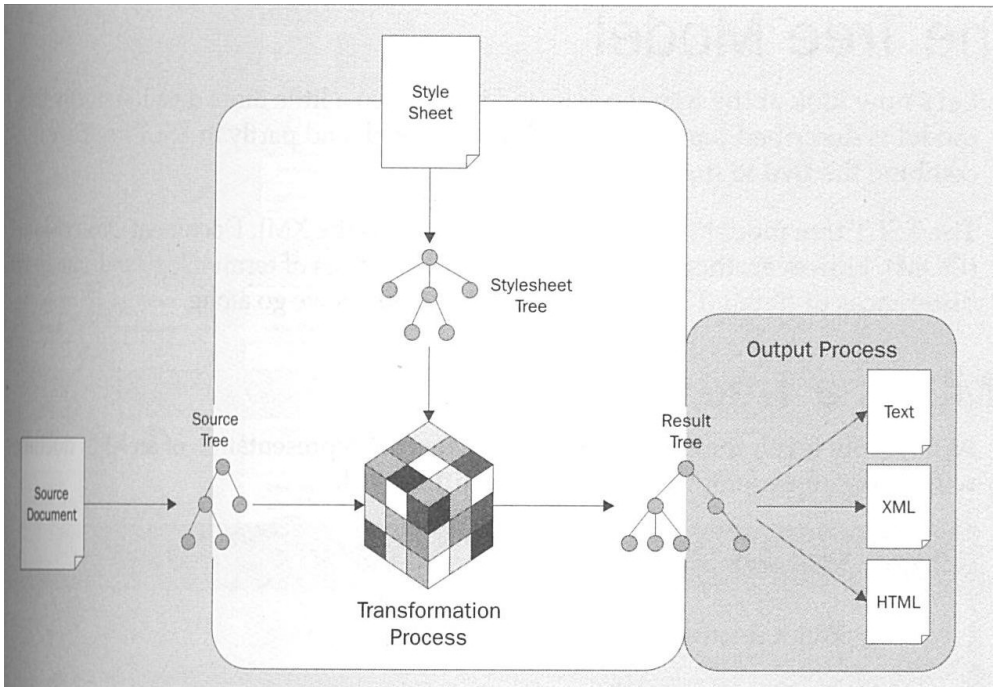


FIGURE 4 FROM XML TO XML, TEXT OR WHAT ELSE

Besides just selecting the presentation, we can also sort them. We can do that with embedding a sort instruction in an `<xsl:apply-templates>` instruction:

```
<xsl:apply-templates select="day/item/presentation">
  <xsl:sort select="author"/>
</xsl:apply-templates>
```

If you want to learn more about XSLT, I can recommend "XSLT Programmer's Reference" by Michael Kay, also the author of the well–known XSLT processor Saxon. For this document I used Xalan, another well–known processor, see `http://xml.apache.org/xalan-c/index.html`.

CONCLUSION

My goal has been to give you a quite exhaustive overview of typesetting structured data, but not already expressed as TEX macro's, with ConTEXt. I did this by showing how you can typesetting XML in ConTEXt. And I covered converting from comma separated files, relational database data and XML to an XML format that can be handled easily by ConTEXt.

♢ ♢ ♢

Usage of MathML for paper and web publishing

Tobias Burnus*

The Mathematical Meta Language (MathML) of the World Wide Web Consortium (W3C) based on XML has gained more support in the last months. Looking at the W3C's list of software which supports MathML one sees that the number of applications which can produce MathML is rather long, but the list of applications supporting typesetting of MathML is rather short.

I will concentrate on those points:

1. Using MathML to write real formulas. I started using it for writing my formulas as a physicist, but I will also use some more complicated examples from the field of physics and mathematics trying to reach the limits of the language.
2. Typesetting MathML on paper in high quality. Writing MathML alone doesn't help if you cannot print it. I will look at the quality of output and alternative representations using ConTeXt.
3. Typesetting on the Web. Except for the fact that there are some applications which can produce MathML and not TEX output, the real use for MathML is the direct and fast representation on the Web. For that I will look at the MathML features of Mozilla.

*Email: burnus@net-b.de

◊ ◊ ◊

The Euromath System – a structured XML editor and browser

J. Chlebíková, J. Guričan, M. Nagy, I. Odrobina
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

abstract. The Euromath System is an XML WYSIWYG structured editor and browser with the possibility of TeX input/output. It was developed within the Euromath Project and funded through the SCIENCE programme of the European Commission. Originally, the core of the Euromath System was based on the commercial SGML structured editor Grif. At present, the Euromath System is in the final stage of re-implementation based upon the public domain structured editor Thot and XML. During the re-implementation process several principal differences between the basic features of Thot and the basic purposes of the Euromath System had to be resolved.

keywords: structured editing, TEX, XML

Introduction

During the last decade we have seen a revolution in the field of processing of the electronic documents. The rapid growth of information technologies brought significant changes to the potential benefits of electronic documents. The current evolution and the research of electronic documents has several basic goals:

- The document can be used for multiple purposes with different applications. Once a document has been stored in electronic form, one should be able to derive multiple "products" from a single source. For example, various kinds of printed material can be produced, the document can be used on the WWW, the document can be searched by database applications, and some parts of the document can communicate with external applications.
- The document has to have a long life-time. It should be easily revised and usable in any stage of its lifetime.
- Documents should be easily interchangeable across different computer platforms and networks.

In line with the previous goals, the markup of documents was developed. For multipurpose documents it is necessary to use general standardized markup with emphasis on

the logical structure of the document. For this reason the idea of a DTD (Document Type Definition) and a syntax taken from GenCode and GML were formalized and SGML (Standard Generalized Markup Language – ISO Standard 8879:1986) was developed (see [4]). Undoubtedly the most popular DTD is HTML, the present language of the WWW.

SGML is a complex standard, and its use remained limited mainly to large companies and a few research institutes. But this is not true for XML (Extensive Markup Language) — the new language of the WWW. The most important difference between XML and HTML is that XML does not have a fixed set of elements and attributes.

Structured Editors

From the user's point of view the most comfortable tool for editing XML documents is a structured editor. A structured editor can guide the user according to the logical structure of the edited document (without his exact knowledge of that structure). In particular, a structured editor can prevent the user from producing a document whose actual logical structure is not consistent with the intended logical structure.

Advanced structured editors can allow the user to deal with the whole logical parts of the document in several ways:

- ◇ move or copy complete logical parts of the document,
- ◇ change an element to an element of another type,
- ◇ create or delete some additional structure around an element
- ◇ and so on.

It is not assumed that the author is familiar with the logical structure of the document. The editor offers options to the user according to the logical structure of the document.

Some structured editors can display and simultaneously allow one to edit individual elements of the logical structure in separate windows, for example the bibliography. It is possible to search for references to a particular logical element, and the editor may facilitate searching for logical elements of a specified type, e.g. tables or figures.

WYSIWYG structured editors have become the most popular; in these the presentation of a document is configured separately for each available logical structure. This approach uses a similar philosophy as LaTeX classes and has several advantages for the user. The author of the document only has to take care of the content of the document — the layout is produced automatically. Also several different presentations can be defined for one logical structure. The editor takes care of updating the numbering of theorems, footnotes, cross-references, etc. according to the logical structure of the document.

At present there are a few freely available WYSIWYG structured editors. We mention Thot, Amaya and the new version of the Euromath System presented here.

Thot ([5]) is an open experimental authoring system developed by the Opera project in order to validate the concepts of a structured document. Thot uses three different internal languages S, P and T for the manipulation of the document, but unfortunately it stores documents in the binary PIV format. From an abstract point of view the

S language (for Structure) provides the same structural concepts as a DTD. The P language (for Presentation) of Thot provides presentation or style sheet support to facilitate WYSIWYG views of documents. The T language (for Translation) allows one to define export specifications for each element (or rules for 'Save As' formats).

Amaya ([3]) is the W3C test-bed browser and authoring tool for HTML documents developed on top of the Thot technology. Amaya also has support for MathML (Mathematical Markup Language) and CSS (Cascading Style Sheet).

In the following we introduce the new version of the *Euromath System* — an XML authoring tool and browser based on Thot. It was developed in the Euromath Support Center (Faculty of Mathematics, Physics and Informatics) in Bratislava.

Euromath System

The primary purpose of the Euromath System is to create a homogeneous computer working environment for mathematicians based on a uniform data model. The first version of the Euromath System (1992) was developed within the Euromath Project led by the European Mathematical Trust. Now the Euromath System combines the advantages of the WYSIWYG approach, structured editing and standardized XML format.

Originally, the core of the Euromath system was based on the commercial SGML structured editor Grif. At present, the core of the Euromath System is in the final step of re-implementation based on XML and Thot. Thot, unlike Grif, is public domain software which is also available for more platforms (Linux, Unix). Due to the conceptual proximity of both editors, the re-implementation was possible.

It is important to say that during the re-implementation process several principal differences between the basic features of Thot and the basic purposes of the Euromath System had to be resolved.

(i) *There is no direct support of XML in Thot.* The problem was solved by a new tool named `DTD2SPT` that is capable of porting an arbitrary XML DTD to the Euromath System. According to the DTD and a feature file the tool generates three files in the internal languages of Thot S, P and T. The S-file describes the logical structure and follows directly from DTD. The P-file is an automatically generated standard non-WYSIWYG 'XML' presentation, in which the logical tree structure of the document is displayed together with the tags for logical elements (which are usually hidden in other presentations). The T-file is used for saving the document in XML format. The user can influence the automatically generated S, P and T-files via certain rules in the feature file. These generated files customize Thot in such a way that it provides comfortable editing for documents which follow the rules of the given DTD. More detailed information about this tool can be found in Subsection .

(ii) *Thot uses the binary PIV format for saving documents.* Therefore, it was necessary to solve the problem of opening and saving XML documents. Due to the fact that Thot has the possibility to add one's own T-language for export of the document, the latter problem was solved almost directly. The problem was solved by adding a mechanism for opening XML documents. It mainly involves the translation of the

input XML document into an internal S-structure according the document's DTD. This is one of the most important features of the Euromath System. Some key issues of the realization of the programme are given in Subsection .

(iii) The third important change follows from the fact that Thot is an authoring system, but the Euromath System is also a WWW browser.

The Euromath system as structured WYSIWYG XML editor

The Euromath System offers the same basic editing functions as non-structured text editors (find-replace, operations with clipboard, ...), and a spelling checker for English, French and other languages. It also offers the possibility to easily incorporate graphics according to various standards. The Euromath System allows easy WYSIWYG addition of new characters with support for UNICODE or the entity mechanism.

The Euromath System enables WYSIWYG structured editing for those DTD's that are frequently used by mathematicians: EmArticle.dtd, EmLetter.dtd, EmSheet.dtd, EmSlide.dtd, and others. These DTD's correspond to LATEX classes. As was mentioned before, the Euromath System facilitates structured editing of documents according any new DTD. But this requires that one write a presentation of the new DTD in Thot's P-language. Moreover, the user can add more than one presentation, so that one eventually has several different presentations for one DTD.

Furthermore, the Euromath System enables to use the advantages of the modularity approach. The user can defined some parts of the document structure as a modul, for example a table, etc. It is comfortable to use modularity approach in the case, if the parts of the document structure are identical for some document classes. Euromath System contain the moduls for paragraphs, tables and mathematical expressions (WYSIWYG presentations, saving into LATEX, ...). From these moduls the WYSIWYG presentation of new documents can be created rather straightforwardly. If the document consists of modules, the Euromath System allows one to change the presentation for each module during editing.

After re-implementation to the structured editor Thot, the Euromath System offers the same properties of the structured editor as before (see [1] for overview). The most of them was mention in the Section .

The Euromath System stores a document automatically in XML format according the corresponding DTD. In the T-language the user has the possibility to add a translation of a document to other structured formats (for example, transform the structure to another one) and to unstructured formats like TEX or HTML. As the system deals with a structured document, the translation to most formats is easy. But for a perfect translation to TEX the T-language lacks certain features (more conditions, external files, etc.).

The Euromath System offers export of documents according to the EmArticle and EmLetter DTDs to the standard LATEX classes article, letter and book. For a new document class the translation to LATEX can easily be built from the available modules such as paragraphs, tables, formulae, etc.

Euromath Applications

The Euromath System comes with a programming interface that allows external actions to be attached to it. Euromath applications extend the possibilities of the Euromath System as a structured editor.

Personal File System (PFS)
The Personal File System is a front-end for the ZentrallBlatt Math database. The form for formulating a (database) query is part of the Euromath System and uses the internal Thot library.

PFS is technically based on three external programs — `pcmes`, `zbl2tex` and `l2s`. `pcmes` comes from ZentrallBlatt, `zbl2tex` and `l2s` are part of the Euromath System and were developed in the framework of the Euromath Project.

`pcmes` is a database engine which executes queries formulated by the user using the PFS form, and generates record sets or other results. These results are returned to the Euromath System using a special listener.

`zbl2tex` transforms a record set obtained from pcmes to an XML file containing the required bibliographic data. This file is opened as a new document using the EmArticle DTD.

Some parts of a database record (TI-title, AB-abstract and UT- keywords) can contain TEX expressions and therefore must be processed by l2s. The source part of the record, which contains bibliographic data, does not have a fixed structure. Especially, books and conference articles can be very complicated and differ from case to case. We use a few heuristic methods to transform this part to the XML structure specified by the EmArticle DTD. These heuristics successfully cover more than 95 %. To get the resulting XML file, `zbl2tex` parses the original output from `pcmes` together with some auxiliary data in three stages.

Translation from TEX to XML – `l2s`
`l2s`, written for the Euromath project, is a parametrizable translation program, which translates TEX, LATEX and AMS-TEX files to files in XML format following the rules of EmArticle DTD (MathML DTD is in progress). Both the lexer and the parser parts contain some new features in order to cope e.g. with default parameters in LATEX2e macros, with the `ensuremath` construction or with `proclaim` in the amsppt style. l2s was mainly written for the translation of LATEX(2e) article style formatted documents to XML format.

The main part of `l2s` is also used for another way of inputting mathematics. The user can insert mathematics using either the standard means of a structured editor or by inserting TEX expressions as special elements. The translation between the TEX formulae and the XML structure can be done at any time by pressing `Ctrl-T` (or using the menu). Even when writing a text, one can press `Ctrl-T` to obtain the possibility to enter TEX formulae directly. When done, pressing `Ctrl-T` again displays the formulae in WYSIWYG mode.

The user can use his own predefined set of TEX macros given in a file determined by the MFILEL2S environment variable.

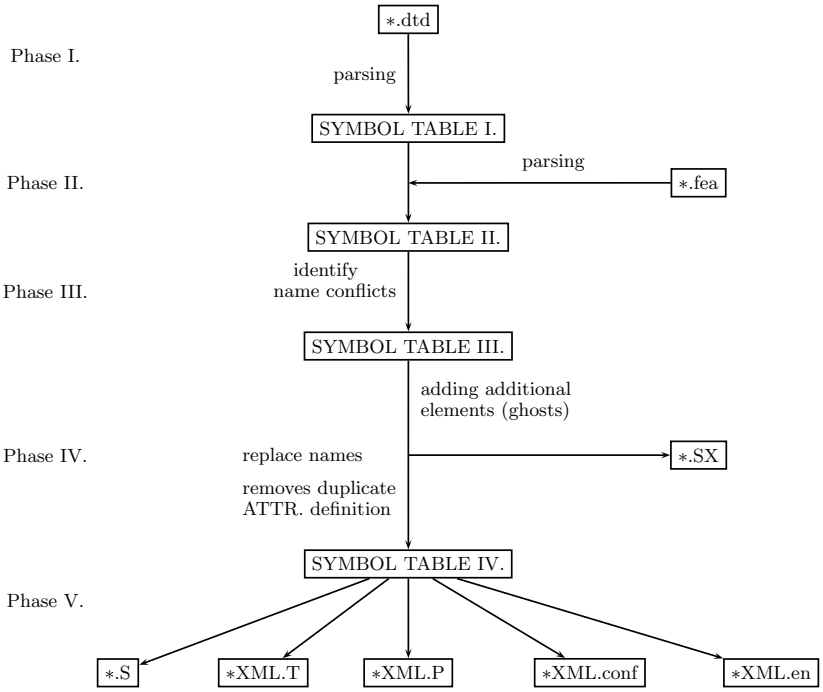


FIGURE 1: THE PHASES AND ACTIONS OF THE DTD2SPT. THE SX FILE IS AN AUXILIARY FILE DESCRIBING THE NAME CONVERSIONS OF THE OBJECTS.

Retrieving documents across networks

The Euromath System can be used as a WWW browser for HTML and XML documents. The Euromath System is ideal for viewing remote XML files – especially for documents with DTD's, for which WYSIWYG presentations in the Euromath System are available. The EmArticle DTD also contains some basic mathematical constructions but in the future we would like to add a WYSIWYG presentation for standard MathML.

IMPLEMENTATION OF THE PROGRAMME

The DTD2SPT tool for porting a DTD into the Euromath System

The tool translates a DTD to files in the S, P, T languages, which are accepted by Thot. The translation process, inputs, outputs and internal states are shown in Figure . The tool passes through five phases, which are separated by four defined states of the principal internal data structure called the SYMBOL TABLE.

During phase I `DTD2SPT` reads the DTD and transforms it into its internal data structure. The user might find it useful to influence the automatic generation of the S, P, T language files by slightly changing the original DTD. This is enabled via

supplemental commands in the *feature file*. As can be seen in Figure , the DTD2SPT alters the internal representation of the DTD. This process is carried out in phase II and the SYMBOL TABLE is transformed to its second state.

During phase III DTD2SPT identifies those names of DTD objects (elements, attributes, attribute-definition-parts) which cannot be used in the S, P, T languages. This happens because the DTD specification permits names with a larger number of characters from a broader set; the DTD-object names may also interfere with keywords of the S, P, T languages.

In addition to these syntactical limitations, we had to deal with a dissimilarity in the background model of objects in the DTD and in the S, P, T languages. An attribute in the DTD is a sub-part of a single element, and the attribute-definition-part is a subpart of an attribute. On the other hand, an attribute in the S-language is visible to all elements, and an attribute-definition-part to all attributes. This clearly implies that, unlike in DTDs, the names of all S-objects must be unique. Due to this dissimilarity, a straightforward translation of a DTD might also generate several definitions of a uniquely named attribute. Therefore during phase IV, attribute names and attribute definitions are compared; conflicts in definitions are resolved and redundant definitions are removed.

The content model of a DTD element can contain also group of elements or occurrence indicators. Here we encounter another problem that prevents a straightforward translation because generation of the presentation (the P-language) for this type of elements represents a serious complication. The problem is resolved during the same translation phase when these multi- composed elements are disassembled into elements with a simple structure. This process introduces additional elements (called ghosts) into the generated S-code.

Opening XML documents

The XML parser is the basic component of the Euromath System. The chosen approach uses the advantages of RTN (recursive transition network). The idea was adapted from [2], who had used RTN to parse natural language sentences. Allen's design was changed in detail to simplify the implementation.

The context-free grammar described in [6] was changed to RTN diagrams. Mainly, for each grammar rule one RTN was designed. The transition arc is labeled either by the terminal string or by a nonterminal one, which represents another RTN. An RTN is implemented as one function, which returns `accept`, `reject` or `error message`. When an RTN returns reject, it must restore the position of the read head. Problems could occur in the case of inclusions (internal or external entities) because it is time consuming to restore the original state before calling the RTN function. But the grammar rules in [6] are well proposed and the grammar is unambiguous. The unambiguity ensures that an RTN can decide whether to return reject or to continue processing (after which it can return only accept or error) by reading a few terminals from the tape.

Semantic rules have been implemented into the RTN functions. For example, the same name of the start-tag and the end-tag, the inclusion of entities, management of processing instructions, ... The inner structure of the document is built while the

input document is read with the assistance of Thot in accordance with RTN parsing. The validity of the document is checked automatically by the Thot engine according the internal structure of the S-language.

In order to successfully open XML documents, it was necessary to make some changes to Thot's S-language. For example, we introduced string conversion, new types of elements and so on.

Unicode has been a serious problem and it has not yet been fully resolved. The Thot engine does supports not it at all. In the present version, the large unicode numbers may be used as well a character entity references. Predefined named unicode entities can be automatically included into every document.

Changed presentation features of the document (for example, the font name or the font size) and the name of the actual presentation are stored as processing instructions.

Concluding Remarks

XML is rapidly becoming the standard for the WWW. The Euromath System is at the forefront in exploiting the benefits of XML for documents and also the typesetting qualities of the TEX system.

The latest version of the Euromath system is available for UNIX (X-window system) on the SUN platform and Linux.

More information about the Euromath system can be found on the page `http:\\www.dcs.fmph.uniba.sk\~emt`.

Our further plans involve support of MathML, improving the possibility of structural changes and supporting at least some feature of CSS or XSL.

REFERENCES

[1] J. Chlebíková, The Euromath System – the structured editor for mathematicians, EuroTeX 1998, pp. 82–93.

[2] J. Allen, Natural Language Understanding, 2nd ed., The Benjamin/Cummings Publishing Company, Inc., CA, 1995.

[3] V. Quint, I. Vatton, An Introduction to Amaya, World Wide Web Journal, Vol. 2, Num. 2, 39–46, 1997.

[4] International Standard ISO 8879, Information Processing – Text and Office Systems – Standard Generalized Markup Language, International Standard Organization, 1986.

[5] Opéra, Thot, A structured document editor, Inria, 1997. `http://www.inrialpes.fr/opera/Thot.en.html`

[6] XML specification, version 1.0, see `http://www.w3c.org/TR/REC-xml`.

♢ ♢ ♢

Instant Preview and the TEX daemon

Jonathan Fine

ABSTRACT.
Instant Preview is a new package, for use with Emacs and `xdvi`, that allows the user to preview instantly the file being edited. At normal typing speed, and on a 225MHz machine, it refreshes the preview screen with every keystroke.
Instant Preview uses a new program, `dvichop`, that allows TEX to process small files over 20 times quicker than usual. It avoids the overhead of starting TEX. This combination of TEX and `dvichop` is the TEX daemon.
One instance of the TEX daemon can serve many programs. It can make TEX available as a callable function. It can be used as the formatting engine of a WYSIWYG editor.
This talk will demonstrate Instant Preview, describe its implementation, discuss its use with LATEX, sketch the architecture of a WYSIWYG TEX, and call for volunteers to take the project forward.
Instant Preview at present is known to run only under GNU/Linux, and is released under the GPL. It is available at:
`http://www.activetex.org`

Instant Preview

TEX is traditionally thought of as a batch program that converts text files into typeset pages. This article describes an add-on for TEX, that in favourable circumstances can compile a file in a twentieth of the normal time. This allows TEX to be used in interactive programs. This section describes Instant Preview.[1]

Types of users

Almost all users of TEX are familiar with the edit-compile-preview cycle that is part of the customary way of using TEX. Previewing is very useful. It helps avoid wasting paper, and it saves time. In the early days, it could take several seconds to compile and preview a file, and perhaps minutes to print it. Today it takes perhaps about a quarter of a second to compile and preview a file.

[1] A screen-shot, in PNG format, is available at the author's website. [It would be nice if it could be included, but I don't know how to do this.]

Many of today's newcomers to computing, and most users of WYSIWYG word processors, expect to have instant feedback, when they are editing a document. Users of TEX expect the same instant feedback, when they are editing a source file in a text editor. Because they have absorbed the meaning of the markup codes, they can usually imagine without difficulty the printed form of the document. They know when the markup is right.

Beginners tend to compile the document frequently, because they are uncertain, and wish to have the positive reinforcement of success. Instant Preview, again under favourable circumstances, can reduce to a twentieth the time take to compile and preview a file. This makes it practical to offer preview after every keystroke. Beginners will be able to see their failures and successes as they happen.

Experienced users do not need such a high level of feedback, and prefer to devote the whole screen to the document being edited. However, even experts have the same need for positive reinforcement, when they use a package that is new to them.

Modus operandi

Here we describe three possible ways of using Instant Preview. At the time of writing, only the last has been implemented. We assume that the document is in the editing stage of its life cycle, or in other words the location of page breaks and the like is not of interest.

The expert needs only occasionally to preview the source document. She will select the region of interest, and ask for it to be previewed. Instant Preview here may provide a quick and convenient interface, but the operation is uncommon and so the functionality should be unobtrusive.

When doing something tricky, the user might wish to focus on a part of the document, and for this part have Instant Preview after every keystroke. The tuning of math spacing in a formula is an example. Few if any users invariably know, without looking, what tuning should be applied to a moderately complicated formula. This applies particularly to displayed equations wider than the measure, multi-line equations, and commutative diagrams. It also applies to the picture environment (for which the special tool TEXcad was written).

For the beginner, everything is tricky, even straight text. The beginner hardly knows that `\{}#^_%$` are all special characters, and that ‘‘ and ’’ are the way to get open and close double quotes. Even experts, who know full well the rules for spaces after control sequences, sometimes make a mistake[2]. The absolute beginner is likely to want Instant Preview of everything, absolutely all the time. Later, with experience, the training wheels can be removed.

Implementation

Instant Preview has been implemented using Emacs and `xdvi`. There seems to be no reason why another editor and previewer should not be used, provided the editor is sufficiently programmable, and the previewer can be told to refresh the file it is previewing.

[2]In the first draft, the allegedly expert author forgot that `&` is also special, and also that `\verb` cannot be used in a LATEX footnote.

Instant Preview works by writing the region to be previewed, together with suitable preamble and postamble, to a special place. From there, the TEX daemon picks it up, typesets it, and writes it out as a `dvi` file. Once written, the previewer is told to refresh its view of the `dvi` file.

The main difference between the three modes is what is written out, and when. Absolute beginner mode writes out the whole buffer, after every keystroke. Confident expert mode writes out a selected region, but only on demand.

At the time of writing (mid-June 2001), only absolute beginner mode has been implemented. Further progress requires above all clear goals and Emacs programming skills.

The dvichop program

For interactive programs, speed is of the essence. Therefore, we will look at TEX's performance. The author's computer has a 225MHz Cyrix CPU. So that we have a definite figure, we will say that on this machine a response time of 1/10 seconds is acceptable.

Typesetting `story.tex`

There is a file, `story.tex`, that is part of every TEX distribution. It is described in *The TEXbook*. On the author's computer, the command

```
time tex ./story \\end
```

takes .245 seconds to execute[3]. This seems to make Instant Preview impossible.

However, the command

```
time tex \\end
```

takes only .240 seconds to execute. Therefore, it takes TEX almost 1/4 of a second to load and exit, while typesetting the two short paragraphs in `story.tex` can be done about 20 times in the target time of a tenth of a second.

Thus, provided the overhead of loading (and exiting) TEX can be avoided, Instant Preview is possible.

Remarks on performance

The simple tests earlier in this article show that it takes TEX about 0.005 seconds to typeset the file `story.tex`. This subsection gives a more precise result. It also show some of the factors that can influence apparent performance.

The file `100story.tex` is as below.

```
\def\0{\input ./story }
\def\1{\0\0\0\0\0\0\0\0\0\0}
\def\2{\1\1\1\1\1\1\1\1\1\1}
\2 \end
```

[3]To avoid the overhead of X-windows, this command was executed in a virtual console. The same goes for the other timing data. The input file is placed in the current directory to reduce kpathsea overheads.

Mode	seconds
Console, output to `/dev/null`	.492
Console, output to screen	.507
X-windows, output to `/dev/null`	.497
X-windows, output to screen	.837

TABLE 1: TIME TAKEN TO TYPESET `STORY.TEX` 100 TIMES

Table 1 gives the time taken to process this file, in the various modes. It shows that on the author's machine and in the best conditions, it takes about $0.0025 \approx (0.492 - 0.240)/100$ seconds to process `story.tex` once.

Note that the time taken can be quite sensitive to the mode, particularly X-windows. We also note that using `\input story` (so that kpathsea looks for the file) adds about 0.025 seconds to the total time taken.

Starting TEX once

The solution is to start TEX once, and use it to typeset multiple documents. Once TEX has typeset a page, it uses the `\shipout` command to write it to the `dvi` file. The new page now exists on the file system, and can be used by other programs. Actually, this is not always true. To improve performance, the system dependent part of TEX usually buffers the output `dvi` stream. However, this can be turned off. We assume that `dvi` output is unbuffered.

Most dvi-reading applications are unable to process such an ill-formed `dvi` file. For example, most immediately seek to the end of the file, to obtain a list of fonts used. To bridge this gap, and thereby enable Instant Preview, the author wrote a utility program called `dvichop`.

This program takes as input a `dvi` file, perhaps of thousands of pages, and produces from it perhaps thousands of tiny `dvi` files. The little files are the ones that the previewer is asked to reload.

More exactly, `dvichop` looks for special *marker pages* in the output dvi-stream produced by TEX the program. The marker pages delimit the material that is to be written to the small `dvi` files. The marker pages also control where the output of `dvichop` is to be written, and which process is to be informed once the output page is ready.

Implementation

The program `dvichop` is written in the *C* programming language. It occupies about 800 lines of code, and calls in a header file `dviop.h` to define the opcodes. A shell program `texd` starts TEX and sends its `dvi` output to `dvichop`. More exactly, TEX writes to a named pipe (a FIFO), which is then read by `dvichop`.

More on performance

In the abstract is is claimed that TEX together with `dvichop` is over 20 times quicker that ordinary TEX, when applied to small files. Here is some test data to support this bold claim.

Normally, `dvichop` is run using a pipe. To simplify matters, we will create the input stream as a ordinary file. The plain input file listed below does this. It also illustrates the interface to `dvichop`.

```
% 100chop.tex
\newcount\dvicount
\def\0{
\begingroup % begin chop marker page
  \global\advance\dvicount 1
  \count0\maxdimen \count1 3
  \count2 \dvicount \shipout\hbox{}
\endgroup
\input ./story % typeset the story
\begingroup % end chop marker page
  \count0\maxdimen \count1 4
  \count2 0 \shipout\hbox{}
\endgroup
}
\def\1{\0\0\0\0\0\0\0\0\0\0}
\def\2{\1\1\1\1\1\1\1\1\1\1}
\begingroup % say hello to dvichop
  \count0\maxdimen \count1 1
  \count2 1 \shipout\hbox{}
\endgroup
\2 % ask dvichop to produce 100 files
\begingroup % say goodbye to dvichop
  \count0\maxdimen \count1 2
  \count2 0 \shipout\hbox{}
\endgroup
\end
```

Typesetting `story.tex` 100 times in the conventional way takes approximately 24.5 seconds. Running TEX on `100chop.tex` takes about 0.510 seconds. This typesets the story for us 100 times. Running `dvichop` on the output file `100chop.dvi` takes 0.135 seconds. Its execution creates files `1.dvi` through to `100.dvi` that are for practical purposes identical to those obtained in the conventional way. The conventional route takes 24.5 seconds. The `dvichop` route took $0.510 + 0.135 = 0.645$ seconds.

This indicates that on `story.tex` using `dvichop` is $24.5/0.635 \approx 38$ times quicker. Some qualifying remarks are in order. In practice, using the pipeline will add overhead, but this seems to be less than 0.01 seconds. On the other hand, the present version of `dvichop` is not optimised.

The TEX daemon

A this point we assume the reader has some basic familiarity with client-server architecture. A server is a program that is running more or less continually, waiting for

requests from clients. Clients can come and go, but servers are expected to persist. An operating system is a classic example of a server, while an application is a client.

Thanks for the memory
Normally, TEX is run as an application or client program. It is loaded into memory to do its job, it does its job, and then it exits. In the mid-1980s, when the author started using a personal computer, having more than a megabyte of memory was uncommon. TEX is uncomfortable on less than 512Kb of memory. Thus running TEX as a server would consume perhaps half of the available memory. For all but the most rabid TEX-ophile, this is clearly not an option.

Today TEX requires perhaps 2Mb of memory, and personal computers typically have at least 32Mb of memory. Letting TEX remain in memory on a more or less permanent basis, much as Emacs and other programs remain loaded even when not used, is clearly practical. However, even today, for most users there is probably not room to have more than a handful of instances of TEX resident in memory.

Sockets
The present implementation of Instant Preview uses a named pipe. Sockets provide a more reliable and flexible interface. In particular, sockets can handle contention (requests to the same server from several clients). Applications communicate to the X-server provided by X-windows by means of a socket.

Providing a socket interface to the TEX daemon will greatly increase its usefulness. The author hopes that by the end of the year he or someone else will have done this.

TEX as a callable function
Over the years, many people have complained that the batch nature of TEX makes it unsuitable for today's new computing world. They have wanted TEX to be a callable function. However, to make TEX a callable function, all that is required is a suitable wrapper, that communicates with the TEX daemon.

At present the TEX daemon is capable of returning only a `dvi` file. To do this, it must parse the output `dvi` stream. Suppose, for example, that the caller wants to convert the output `dvi` into a bitmap, say for inclusion in an HTML page. The present set-up would result in the `dvi` pages being parsed twice. Although this is not expensive, compared to starting up a whole new TEX process, it is still far from optimal.

If the TEX daemon could be made to load page-handling modules, then the calling function could then ask for the bitmap conversion module to handle the pages produced by the function call. This would be more efficient. However, as we shall soon see, premature optimisation can be a source of problems.

TEX forever
An errant application does not bring down the operating system. Strange keystrokes and mouse movements do not freeze X-windows. In the same way, applications should never be able to kill the TEX daemon. To achieve this level of reliability is something of a programming problem.

One thing is clear: The application cannot be allowed to send arbitrary raw TEX to the TEX daemon. TEX is much too sensitive. All it takes is something like

```
\global\let\def\undefined
```

and the TEX daemon will be rendered useless.

A more subtle form of this problem is when a client's call to the daemon results in an unintended, unwelcome, and not readily reversible change of state. For example, the LATEX macro `\maketitle` executes

```
\global\let\maketitle\relax
```

which is an example of such a command. (Doing this frees tokens from TEX's main memory. When TEX, macros and all, is shoe-horned into 512Kb, this may be a good idea.)

Protecting TEX

TEX can be made a callable function by providing an interface to the TEX daemon. Most applications will want an interface that is safe to use. In other words, input syntax errors are reported before they get to TEX, and it is not possible to accidentally kill the TEX daemon. To provide this, the interface must be well defined. For example, the input might be an XML-document (say as a string) together with style parameters, and the output would be say a `dvi` file. Alternatively, the input might be a pointer to an already parsed data structure.

In the long run, this interface is probably best implemented using compiled code, rather than TEX macros. Once a function is used to translate source document into TEX input, there is far less need for developers to write complicated macros whose main purpose is to provide users with a comfortable input syntax. Instead, the interface function can do this.

When carried out in a systematic manner, this will remove the problem, that in general LATEX is the only program that can understand a LATEX input file. The same holds for other TEX macros formats, of course. Note that Don Knuth's `WEAVE` (part of his literate programming system) is similarly compiled code that avoids the need to write complicated TEX macros.

Visual TEX

This article uses the term visual TEXto mean programs and other resources that allow the user to interact with a document through a formatted representation, typically a previewed `dvi` file. We use it in preference to WYSIWYG (what you see is what you get) for two reasons. The first is today many documents are formatted only for screen, and never get printed. Help files and web pages are examples of this. The second is that even when editing a document for print, the user may prefer a representation that is not WYSIWYG.

In most cases the author will benefit from interacting with a suitably formatted view of the underlying document. The benefits of readability and use of space that typesetting provides in print also manifest on the screen. But to insist on WYSIWYG

is to ignore the differences between the two media. Hence our use of the term Visual TEX.

Whatever term is used, the technical problems are much the same, which is how to enable user interaction with the `dvi` file.

Richer `dvi` files

In Visual TEX, the resulting `dvi` file is a view on the underlying document. For it to be possible to edit the document through the view, the view must allow the access to the underlying document. Editing changes applied to the view, such as insertion and deletion, can then be applied to the document.

Placing large numbers of `\special` commands in the `dvi` file is probably the best (and perhaps the only) way to make this work. Doing this is the responsibility of the macro package (here taken to include the input filter function described in the previous section). It is unlikely that any existing macro package, used in its intended manner, will support the generation of such enriched `dvi` files. The author's Active TEX macro package[2] is designed to allow this.

Better `dvi` previewers

Most `dvi` previewers convert the `dvi` into a graphics file, such as a bitmap. Some retain information about the font and position of each glyph. A text editor or word processor has a cursor (called point in Emacs), and by moving the cursor text can be marked. This is a basic property of such programs. So far as the author knows, no `dvi` previewer allows such marking of text.

Further reading

This section is based on the author's article [1].

The Lyx editor for LATEX adopts a visual approach to the generation of files that can be typeset using LATEX. It does not support WYSIWYG interaction. Understanding the capabilities and limitations of Lyx is probably a good way to learn more about this area.

The next steps

This section discusses some of the opportunities and problems in this general area, likely to present themselves over the next year or two.

Applications

Two areas are likely to be the focus of development in the next year or so. The first is the refinement of Instant Preview, as a tool for use with existing TEX formats. Part of this is the creation of material for interactive (La)TEX training. Instant Preview provides an attractive showcase for the abilities of TEX and its various macro packages.

The second is TEX as a callable function. This is required for Visual TEX. One of the important missing components are libraries that allow rich interaction with `dvi` files. This will lay the foundation for TEX being embedded in desktop applications.

License

The work described this article is at present released under the General Public Licence of the Free Software Foundation (the GPL). Roughly speaking, this means that any derived work that contains say the author's implementation of the TEX daemon must also be released under the GPL.

However, the TEX daemon is the basis for TEX as a callable function, and for good reason library functions are usually released under the Lesser (or Library) General Public Licence (the LGPL), or something similar. This means that the library as is can be linked into proprietary programs, but that any enhancement to the library must be released under the LGPL.

Porting

TEX runs on almost all computers, and where it runs, it gives essentially identical results. The same applies, of course, to TEX macros. By and large, it is desirable that the tools used with TEX run can be made to run identically on all platforms. This is not to say that the special features of any particular platform should be ignored. Nor is it to say that advances (such as Instant Preview itself) should not first manifest on a more suitable platform.

Cross-platform portability is one of the great strengths of TEX. What is desirable is that programs that run with TEX have a similar portability. Many people cannot freely choose their computing platform. If TEX and friends are available everywhere, this make TEX a more attractive choice.

In the 1980s, in the early days of TEX, many pioneers ported TEX to diverse platforms. This work established deep roots that even today continue to nourish the community. Although Instant Preview, even when fully developed, is not on the same scale as TEX, it being ported will similarly nourish the community.

TEX macros

Visual TEX requires a stable TEX daemon, which in turn will require a macro package (or a pre-loaded format). This new use of TEX places new demands on the macros. Here, we include in macros any input filter functions used to protect the TEX daemon from errant applications.

These new demands include protection against change of state, reporting and recovery from errors, ability to typeset document fragments, support for rich `dvi` file, and the ability for a single daemon to support round-robin processing of multiple documents. Once tools are in place, much of the input is likely to be XML, and much of the output will be for screen rather than paper.

The existing macros packages (such as plain, LATEX and ConTEXT) were not written with these new requirements in mind. Although they are useful now, in the longer term it may be better to write a new macro package from scratch, for use in conjunction with suitable input filters.

SUMMARY

By running TEX within a client-server architecture, many of the problems traditionally associated with it are removed. At the same time, new demands are placed on macro packages, device drivers (such as `dvichop` and `xdvi`) and a new category of software, input filters (such as `WEAVE`).

This new architecture allows Instant Preview, and opens the door to Visual TEX. All this is possible without making any changes to TEX the program, other than in the system dependent part.

Don Knuth

In 1990, when he told us [4] that his work on developing TEX had come to an end, Don Knuth went on to say:

> Of course I do not claim to have found the best solution to every problem. I simply claim that it is a great advantage to have a fixed point as a building block. Improved macro packages can be added on the input side; improved device drivers can be added on the output side.

The work described in this article has taken its direction from this statement. One of the most obvious characteristics of today's computer monitors (not to be confused with the chalk monitor in classrooms of old) is their widespread use of colour. TEX is clumsy with colour. TEX was not designed with Visual TEX in mind. However, we still have our hands full making the best of what we have with TEX. If our labours bear fruit, then in time a place and a need for a successor will arise.

Again, this possibility was foretold by Don Knuth [3]:

> Of course I don't mean to imply that all problems of computational typography have been solved. Far from it! There are still countless important issues to be studied, relating especially to the many classes of documents that go far beyond what I ever intended TEX to handle.

REFERENCES

[1] Jonathan Fine, Editing `.dvi` files, or Visual TEX, *TUGboat*, **17** (3) (1996), 255–259.

[2] ———, Active TEX and the `DOT` input syntax, *TUGboat*, **20** (3) (1999), 248–261

[3] Donald E. Knuth, The Errors of TEX, *Software—Practice & Experience*, **19** (1989) 607–685 (reprinted in *Literate Programming*)

[4] ———, The future of TEX and METAFONT, *TUGboat*, **11** (4) (1990), 489 (reprinted in *Digital Typography*)

♢ ♢ ♢

TEX and/or XML*: good, bad and/or ugly*

Hans Hagen*
PRAGMA ADE, 8061 GH Hasselt, The Netherlands

abstract. As a typesetting engine, TEX can work pretty well with structured input. One can build interfaces that are reasonably well to work with and code in. XML on the other hand is purely meant for coding, and the more rigorous scheme prevents errors and makes reuse easy. Contrary to TEX, XML is not equivalent to typesetting, although there are tools (and methods) to easily convert the code into other stuctured code (like HTML) that then can be handled by rendering engines. Should we abandon coding in TEX in favor of XML? Should we abandon typesetting using TEX in favor of real time rendering of relatively simple layout designs? Who are the good and bad guys in that world? And even more importantly: to what extent will document design (and style design) really change?

*E-mail: pragma@wxs.nl URL: www.pragma-ade.com

♢ ♢ ♢

TEX Top Publishing: an overview

Hans Hagen*
PRAGMA ADE, 8061 GH Hasselt, The Netherlands

abstract. TEX is used for producing a broad range of documents: articles, journals, books, and anything you can think of. When TEX came around, it was no big deal to beat most of those day's typesetting programs. But how well does TEX compete today with mainstream Desk Top Publishing programs?

What directions will publishing take and what role can TEX play in the field of typesetting? What are today's publishing demands, what are the strong and what are the weak points of good old TEX, and what can and should we expect from the successors of TEX?

*E-mail: pragma@wxs.nl, url: www.pragma-ade.com

♢ ♢ ♢

ConTEXt Publication Module, The user documentation

Taco Hoekwater

Introduction

This module takes care of references to publications and the typesetting of publication lists, as well as providing an interface between BibTEXand ConTeXt.

This is a preliminary version; changes may be needed or wanted in the near future. In particular, there are some minor issues with the multi-lingual interface that need to be solved.

The bibliographic subsystem consists of the main module `m-bib.tex`; a helper module (`m-list.tex`); four BibTEX styles (`cont-xx.bst`); and an example configuration file (`bibl-apa.tex`) that specifies formatting instructions for the citations and the list of references.

General overview

A typical input file has the following structure:

1. A call to `\usemodule[bib]`.
2. Some optional setup commands for the bibliographic module.
3. A number of definitions of publications to be referenced in the main text of the article. The source of these definitions can be a combination of:
   - an implicit BibTEX-generated BBL file (read at `starttext`)
   - one or more explicit BibTEX-generated BBL files
   - an included definition file in the preamble
   - included macros before `\starttext`

   All of these possibilities will be explained below. For now, it is only important to realize that of all these definitions must be known *before* the first citation in the text.
4. `\starttext`
5. The body text, with a number of `\cite` commands.
6. The list of publications, called using the command `\placepublications` or the command `\completepublications`.
7. `\stoptext`

Setup commands

Bibliographic references tend to use a specific 'style', a collection of rules for the use of `\cite` as well as for the formatting that is applied to the publication list. The CONTEXT bibliographic module allows one to define all of these style options in one single file. Unlike LATEX, his style includes the formatting of the items themselves.

Global settings: `\setuppublications`

The most important user-level command is `\setuppublications`. Most of the options to this command are set by the bibliography style, and should only be overridden with great care, but a few of them are of immediate interest to the user. The command should be given before `\starttext`, and it sets some global information about the bibliographic references used in the document. CONTEXT needs this information in order to function correctly.

```
\setuppublications[..,..=..,..]

autohang        yes no
numbering       yes no short bib
criterium       all cite
sorttype        bbl cite
alternative     text apa
refcommand      author authoryear authoryears key number num page short type year data
```

alternative — This gives the name of a bibliography style.
Currently, there is only one style, which is APA-like, and that style is therefore also the default.

sorttype — How the publications in the final publication list should be sorted. 'cite' means: by the order in which they were first cited in your text. 'bbl' tells the module to keep the relative ordering in which the publication definitions were found.
The current default for apa is 'cite'.

criterium — Whether to list only the referenced publications or all of them.
If this value is 'all', then if 'sorttype' equals 'cite', this means that all referred-to publications are listed before all others, otherwise (if 'sorttype' equals 'bbl') you will just get a typeset version of the used database(s).
The default for apa is 'used'

numbering — Whether or not the publication list should be labelled and if so, how. `yes` uses the item number in the publication list as label. `short` uses the short label. `bib` uses the original number in the BibTEX database as a label. Anything else turns labelling off.
The default for apa is 'no'.

numbercommand	A macro that can be used to typeset the label if numbering is turned on. The default behaviour is to typeset the label as-is, flush left.
autohang	Whether or not the hanging indent should be re-calculated based on the real size of the label. This option only applies if numbering is turned on. The default is 'no'.
refcommand	The default option for `\cite`.

Since most of the options should be set by a bibliography style, the specification of an alternative bibliography style implies that all other arguments in the same command will be ignored. If you want to make minor changes to the bibliography style, do it in two separate commands, like this:

```
\setuppublications[alternative=apa]
\setuppublications[refcommand=author]
```

How the entries are formatted: `\setuppublicationlist`

`\setuppublicationlist[..,..=..,..]`

totalnumber	*text*
samplesize	*text*
editor	\invertedauthor \invertedshortauthor \normalshortauthor \normalauthor
author	\invertedauthor \invertedshortauthor \normalshortauthor \normalauthor
artauthor	\invertedauthor \invertedshortauthor \normalshortauthor \normalauthor
namesep	*text*
lastnamesep	*text*
firstnamesep	*text*
juniorsep	*text*
vonsep	*text*
surnamesep	*text*
..=..	see \setuplist

The list of publications at the end of the article is essentially a normal context 'list' that behaves much like the list that defines the table of contents, with the following changes:

The module defines a few new options. These options are static, they do *not* change to follow the selected context interface.

The first two options provide default widths for 'autohang':

totalnumber	The total number of items in the following list (used for autohang).
samplesize	The longest short label in the list (used for autohang)

All the other extra options are needed to control micro–typesetting features that are buried deep within macros. There is a separate command to handle the larger

layout options (\setuppublicationlayout, explained below), but the options here are the only way to make changes in the formatting used for the names of editors, authors, and article authors.

editor	command to typeset one editor in the publication list.
author	command to typeset one author in the publication list.
artauthor	command to typeset one article author in the publication list.
namesep	the separation between consecutive names (either editors, authors or artauthors).
lastnamesep	the separation before the last name in a list of names.
firstnamesep	the separation following the first-name or inits part of a name in the publication list.
juniorsep	likewise for 'junior'.
vonsep	likewise for 'von'.
surnamesep	likewise for surname.

The commands that are listed as options for 'editor', 'author' and 'artauthor' are predefined macros that control how a single name is typeset. The four supplied macros provide formatting that looks like this:

\invertedauthor	von Hoekwater, jr Taco
\invertedshortauthor	von Hoekwater, jr T
\normalauthor	Taco, von Hoekwater, jr
\normalshortauthor	T, von Hoekwater, jr

As can be seen in the examples, there is a connection between certain styles of displaying a name and the punctuation used. Punctuation in this document has been set up by the 'apa' style, and that style makes sure that \invertedshortauthor looks good, since that is the default command for 'apa' style. (Keep in mind that the comma at the end of the author will be inserted by either 'namesep' or 'lastnamesep'.)

If you are not happy with the predefined macros, you can quite simply redefine one of these macros. They are all simple macros with 5 arguments: firstnames, von-part, surname, inits, junior.

For reference, here is the definition of \normalauthor:

```
\def\normalauthor#1#2#3#4#5%
     {\bibdoifelse{#1}{#1\bibvariant{firstnamesep}}{}%
      \bibdoifelse{#2}{#2\bibvariant{vonsep}}{}%
                      #3\bibvariant{surnamesep}%
      \bibdoifelse{#5}{#5}{}}
```

But commands can be a lot simpler, like this:

```
\def\surnameonly#1#2#3#4#5{#3}
\setuppublicationlist[editor=\surnameonly]
```

The module itself sets some of the normal options to the setup of a list. To ensure a reasonable layout for the reference list, the following are set as a precaution:

variant	Always re-initialized to 'a'. This makes sure that no space is allocated for the page number.
pagenumber	Always re-initialized to 'no'. The list is a bit of a special one, and page numbers don't make much sense. All entries will (currently) have the same page number: the number of the page on which `\placepublications` was called.
criterium	Always set to 'all'. You need this! If you want partial lists, set 'criterium' to 'used', and 'sorttype' to 'cite'. This combination will reset itself after each call to `\placepublications`.

In addition, the following options are initialized depending on the global settings for 'numbering' and 'autohang':

width	Set to the calculated width of the largest label (only if autohang is 'yes').
distance	Set to 0pt (only if autohang is 'yes').
numbercommand	The command given in 'setuppublications' if numbering is turned on, otherwise empty.
textcommand	Set to a macro that outdents the body text if numbering is turned off, otherwise empty.

Setting citation options: `\setupcite`

The `\cite` command has a lot of sub-options, as can be seen above in the setting of 'refcommand'. And even the options have options:

```
\setupcite[..,...,..][..,..=..,..]

...           author authoryear authoryears key number num page short type year data
pubsep        text
lastpubsep    text
inbetween     text
left          text
right         text
compress      yes no
```

Here are the possible keywords:

pubsep	separator between publication references in a `\cite` command.
lastpubsep	same, but for the last publication in the list.
left	left-hand side of a `\cite` (like `[`).
inbetween	the separator between parts of a single citation.
right	right-hand side of a `\cite` (like `]`).
compress	Whether `\cite` should try to compress its argument list. The default is 'yes'.

Not all options apply to all types of `\cite` commands. For example, 'compress' does not apply to the citation list for all options of `\cite`, since sometimes compression does not make sense or is not possible. The 'num' version compresses into a condensed sorted list, and the various 'author' styles try to compress all publications by one author, but e.g. years are never compressed.

Likewise, 'inbetween' only applies to three types: 'authoryear' (a space), 'authoryears' (a comma followed by a space), and 'num' (where it is '–' (an endash), the character used to separate number ranges).

Setting up BibTEX: `\setupbibtex`

BibTEX bibliographic databases are converted into `.bbl` files, and the generated file is just a more TEX-minded representation of the full database(s).

The four `.bst` files do not do any actual formatting on the entries, and they do not subset the database either. Instead, the *entire* database is converted into TEX-parseable records. About the only thing the `.bst` files do is sorting the entries (and BibTEX itself resolves any 'STRING' specifications, of course).

The module will read the created `\jobname.bbl` file and select the parts that are needed for the current article.

```
\setupbibtex[..,..=..,..]

database    file(s)
sort        no author title short
```

database List of bibtex database file names to be used. The module will write a very short `.aux` file instructing BibTEX to create a (possibly very large) `\jobname.bbl` file, that will be `\input` by the module (at `\starttext`).

sort How the publications in the BibTEX database file should be sorted. The default here is 'no' (`cont-no.bst`), meaning no sorting at all. 'author' (`cont-au.bst`) sorts alphabetically on author and within that on year, 'title' (`cont-ti.bst`) sorts alphabetically on title and then on author and year, and 'short' (`cont-ab.bst`) sorts on the short key that is generated by BibTEX.

For now, you need to run BibTEX by hand to create the `\jobname.bbl` file (`texutil` will hopefully do this for you in the future).

You may want to create the `\jobname.bbl` yourself. The `.bbl` syntax is explained below. There is no default database of course, and you do not *have* to use one: it is perfectly OK to just `\input` a file with the bibliographic records, as long as it has the right input syntax. Or even to include the definitions themselves in the preamble of your document.

The most efficient calling order when using BibTEX is:

```
texexec --once myfile
bibtex myfile
texexec myfile
```

Texexec should be smart enough to recognize how many runs are needed in the final part, but it seems it sometimes does one iteration too few. So you might have to call texexec one last time to get the page references correct. Numbered references always require at least one more run than do (author,year) references, because the final number in the reference list is usually not yet known at the moment the `\cite` command is encountered.

Borrowing publications: `\usepublications`

It is also possible to instruct the module to use the bibliographic references belonging to another document. This is done by using the command `\usepublications[files]`, where `files` is a list of other CONTEXT documents (without extension).

```
\usepublications[..,...,..]

...     file(s)
```

To be precise, this command will use the `.bbl` and `.tuo` files from the other document(s), and will therefore not work if these files cannot be found (the `.tuo` file is needed to get correct page references for `\cite[page]`).

CITATIONS

Citations are handled through the `\cite` command.

`\cite` has three basic appearances:

`\cite[keys]`	Executes the style-defined default citation command. This is the preferred way of usage, since some styles might use numeric citations while others might use a variation of the (author,year) style. 'keys' is a list of one of more publication IDs.
`\cite[option][keys]`	The long form, which allows you to manually select the style you want. See below for the list of valid 'option's.
`\cite{keys}`	For compatibility (with existing LATEX .bib databases). Please don't use this form in new documents or databases.

Cite options

Right now, the interesting bits are the keys for the argument of `\startpublication`.

Following is the full list of recognized keywords for `\cite`, with a short explanation where the data comes from. Most of the information that is usable within `\cite` comes from the argument to `\startpublication`. This command is covered in detail below, but here is an example:

```
\startpublication[k=me,
                  t=article,
                  a=Hoekwater,
```

```
                    y=1999,
                    s=TH99,
                    n=1]
...
\stoppublication
```

All of these options are *valid* in all publication styles, since CONTEXT always has the requested information. But not all of these are *sensible* in a particular style. For instance, using numbered references if the list of publications itself is not numbered is not a good idea. Also, some of the keys are somewhat strange and only provided for future extensions.

First, here are the simple ones:

author	(Hoekwater)	(from 'a')
key	[me]	(from 'k')
number	[1]	(from 'n')
short	[TH99]	(from 's')
type	[article]	(from 't')
year	(1999)	(from 'y')

Keep in mind that 'n' is a database sequence number, and not necesarily the same number that is used in the list of publications. For instance, if 'sorttype' is cite, the list will be re-ordered, but the 'n' value will remain the same. To get to the number that is finally used, use

num [1] (this is a reference to the sequence number used in the publication list)

Even if the list of publications is not numbered visually, a number is still available.

Three of the options are combinations:

authoryear	Hoekwater (1999)	(from 'a' and 'y')
authoryears	(Hoekwater, 1999)	(from 'a' and 'y')
data	Hoekwater, T. (To appear). CONTEXT Publication Module, The user documentation. *MAPS*, pages 66–76. This article.	The data content.

And the last one is a page reference to the *first* place where the entry was cited. This is not always the page number in the list of publications: if there was a `\cite[data]` somewhere in the document, that page number will be the number used (as you can see from the example).

page [68] (a page reference)

Placing the list of publications

This is really simple: use `\completepublications` or `\placepublications` at the location in your text where you want the list of publications to appear. As is normal in CONTEXT, `\placepublications` gives you a raw list, and `\completepublications` a list with a heading. The module uses the following defaults for the generated head:

```
\setupheadtext[en][pubs=References]
\setupheadtext[nl][pubs=Literatuur]
\setupheadtext[du][pubs=Literatur]
```

These can be redefined as needed.

The bbl file

A typical bbl file consists of one initial command (`\setuppublicationlist`) that sets some information about the number of entries in the bbl file and the widths of the labels for the list, followed by a number of occurrences of:

```
\startpublication[k=,
                  t=,
                  a=,
                  y=,
                  s=,
                  n=]
...
\stoppublication
```

The full version of `\cite` accepts a number of option keywords, and we saw earlier that the argument of the `\startpublication` command defines most of the items we can make reference to. This section explains the precise syntax for `\startpublication`.

Each single block defines one bibliographic entry. I apologise for the use of single–letter keys, but these have the advantage of being a) short and b) safe w.r.t. the multi-lingual interface.

Each entry becomes one internal TEX command.

`\startpublication[..,..=..,..]`

k	*text*
a	*text*
y	*text*
s	*text*
t	*text*
n	*text*

Here is the full example that has been used throughout this document:

```
\startpublication[k=me,
                  t=article,
                  a=Hoekwater,
                  y=1999,
                  s=TH99,
                  n=1]
\artauthor[]{Taco}[T.]{}{Hoekwater}
\arttitle{\CONTEXT\ Publication Module, The user documententation}
\journal{MAPS}
\pubyear{To appear}
\note{This article}
\pages{66--76}
\stoppublication
```

Defining a publication

Here is the full list of commands that can appear between `\startpublication` and `\stoppublication`. All top-level commands within such a block should be one of the following (if you use other commands, they might be typeset at the beginning of your document or something similar).

Order within an entry is irrelevant, except for the relative order of the three commands that may appear more than once: `\artauthor`, `\author` and `\editor`.

Here is the full list of commands that can be used. Most of these are 'normal' BibTEX field names (in lowercase), but some are extra special, either because they come from non-standard databases that I know of, or because the bst file has preprocessed the contents of the field:

`\abstract#1`	Just text.
`\annotate#1`	Just text.
`\artauthor[#1]#2[#3]#4#5`	For an author of any publication that appears within a larger publication, like an article that appears within a journal or as part of a proceedings.
`\arttitle#1`	The title of such a partial publication.
`\author[#1]#2[#3]#4#5`	The author of a standalone publication, like a monograph.
`\chapter#1`	The chapter number, if this entry refers to a smaller section of a publication. It might actually be a part number or a (sub)section number, but the BibTEX field happens to be called `CHAPTER`. The field `\type` (below) differentiates between these.
`\city#1`	City of publication.
`\comment#1`	Just text.
`\country#1`	ccountry of publication.

`\crossref#1`	A cross-reference to another bibliographic entry. It will insert a citation to that entry, forcing it to be typeset as well.
`\edition#1`	The edition.
`\editor[#1]#2[#3]#4#5`	The editor of e.g. an edited volume.
`\institute#1`	The institute at which the publication was prepared.
`\isbn#1`	isbn number (for books).
`\issn#1`	issn number (for journals).
`\issue#1`	issue number (for journals).
`\journal#1`	The journal's name.
`\keyword#1`	Just text (for use in indices).
`\keywords#1`	Just text (for use in indices).
`\month#1`	Month of publication.
`\names#1`	Just text (for use in indices).
`\note#1`	Just text (this is the 'standard' BibTEX comment field).
`\notes#1`	Just text.
`\organization#1`	Like institute, but e.g. for companies.
`\pages#1`	Either the number of pages, or the page range for a partial publication. The 't' key to startpublication will decide automatically what is meant.
`\pubname#1`	Publisher's name.
`\pubyear#1`	Year of publication. Within this command, the BibTEX bst files will sometimes insert the command `\maybeyear`, which is needed to make sure that the bbl file remains flexible enough to allow all styles of formatting.
`\series#1`	Possible book series information.
`\size#1`	Size in KB of a PDF file (this came from the NTG Maps database).
`\thekey#1`	BibTEX's 'KEY' field. See the BibTEX documentation for its use. This is *not* related to the key used for citing this entry.
`\title#1`	The title of a book.
`\type#1`	BibTEX's 'TYPE' field. See the BibTEX documentation for it's use. This is *not* related to the type of entry that is used for deciding on the layout.
`\volume#1`	Volume number for multi-part books or journals.

Rather a large list, which is caused by the desire to support as many existing BibTEX databases as possible.

As you can see, almost all commands have precisely one argument. The only exceptions are the three commands that deal with names: `\artauthor`, `\author` and `\editor`. At the moment, these three commands require 5 arguments (of which two look like they are optional, they are *not*!)

Adding one of your own fields is reasonably simple:

```
\newbibfield[mycommand]
```

This will define `\mycommand` for use within a publication (plus `\bib@mycommand`, its internal form) as well as the command `\insertmycommand` that can be used within `\setuppublicationlayout` to fetch the supplied value (see below).

Defining a publication type layout

Publication style files of course take care of setting defaults for the commands as explained earlier, but the largest part of a such a publication style is concerned with specifying layouts for various types of publications.

The command that does the work is `\setuppublicationlayout`. It has an optional argument that is a `type`, and all publications that have this type as argument to the 't' key of `\startpublication` will be typeset by executing the commands that appear in the group following the command.

For reference, here is one of the commands from `bibl-apa`:

```
\setuppublicationlayout[article]{%
   \insertartauthors{}{ }{\insertthekey{}{ }{}}%
   \insertpubyear{(}{). }{\unskip.}%
   \insertarttitle{\bgroup }{\egroup. }{}%
   \insertjournal{\bgroup \it}{\egroup}
    {\insertcrossref{In }{}{}}%
   \insertvolume
    {, }
    {\insertissue{(}{)}{}\insertpages{:}{.}{.}}
    {\insertpages{, pages }{.}{.}}%
   \insertnote{ }{.}{}%
   \insertcomment{}{.}{}%
}
```

For every command in the long list given in the previous section, there is a corresponding `\insertxxx` command. (As usual, `\author` etc. are special: they have a macro called `\insertxxxs` instead.) All of these `\insertxxx` macros use the same logic:

```
\insertartauthors{<before>}{<after>}{<not found>}
```

Sounds easy? It is! But it is also often tedious: database entries can be tricky things: some without issue numbers, others without page numbers, some even without authors. So, you often need to nest rather a lot of commands in the `<not found>` section of the 'upper' command, and `\unskip` and `\ignorespaces` are good friends as well.

There is nothing special about the type name you give in the argument, except that every `\startpublication` that does not have a 't' key is assumed to be of type 'article', and undefined 't' values imply that the data is completely ignored.

`bibl-apa` defines layouts for the 'standard' publication types that are defined in the example bibliography that comes with BibTEX.

Bibliography

Hoekwater, T. (To appear). CONTEXT Publication Module, The user documentation. *MAPS*, pages 66–76. This article.

♢ ♢ ♢

MLBIBTEX: *a New Implementation of* BIBTEX

JEAN-MICHEL HUFFLEN*

ABSTRACT. This paper describes MLBIBTEX, a new implementation of BIBTEX with multilingual features. We show how to use it as profitably as possible, and go thoroughly into compatibility between BIBTEX's current implementation and ours. Besides, the precise grammar of MLBIBTEX is given as an annex.

KEYWORDS: Bibliographies, multilingual features, LATEX, BIBTEX.

INTRODUCTION

THERE is increasing interest in multilingual word processors nowadays: some books may be composed of parts written in different languages, some documents have to be produced using several languages: for example, at least three languages for official documents of the EEC[1]. As word processors, TEX and LATEX are indisputably pioneers in this topic. From its first version, TEX [28] has provided commands to produce accents and other diacritical signs of European languages using the Latin alphabet. The ability for non-English words to be hyphenated has been improved by first MLTÊX (for 'Multilingual TÊX') [16] and then TEX's Version 3. When LATEX2ε [32] came out, the french [19] and german [41] packages strongly eased writing documents in French and German, even if these packages were *ad hoc* for one language only. 'Actual' multilinguism has been reached with the babel package [7], in the sense that this package processes all the languages it knows in a homogeneous way, without giving any privilege to a particular one. Therefore this package is especially suitable for mixing several languages within the same document. Besides, this package is now able to process some languages using a non-Latin alphabet (Greek, Russian, ... cf. [7, § 26 & 51]). Last but not least, the Ω and Λ projects [40] aim to develop 'super' TEX and LATEX engines, able to process all the world's languages[2], by using the Unicode standard encoding [45].

Now let us consider BIBTEX [38], the bibliography program associated with LATEX.

*Author's address: LIFC, University of Franche-Comté. 16, route de Gray. 25030 BESANÇON CEDEX. FRANCE. E-mail: hufflen@lifc.univ-fcomte.fr.

[1] **E**uropean **E**conomic **C**ommunity.

[2] In fact, some extensions of TEX were already able to process languages with right-to-left texts as well as languages with left-to-right texts: TEX-XETEX since 1987 [27], and ε-TEX [36].

```
@BOOK{gibson1988,
        AUTHOR = {William Gibson},
        TITLE = {Mona Lisa Overdrive},
        PUBLISHER = {Victor Gollancz, Ltd.},
        YEAR = 1988}
```

Figure 1: Example of a BIBTEX entry.

It offers some flexibility about foreign (that is, non-English) language support (cf. [21, § 13.8.2]), and the insertion of some slight multilingual features have been put into action: for example, the bibliography style files built by means of the makebst program (cf. [13] or [21, § 13.9]) can be interfaced with the babel package; another example is given by the Delphi BibStyles collection (cf. [21, § 13.8.2]). But it is obvious that BIBTEX's present version does not provide as many multilingual features as LATEX's.

Given these considerations, we started a new implementation in October 2000, so-called MLBIBTEX (for 'Multilingual BIBTEX'). The first version (1.1) is available now[3], and has already been introduced in [24], but very informally. For the EUROTEX conference, we propose a more precise description hereafter.

In this article, we do not pay particular attention to the typographical conventions ruling the layout of bibliographies, since MLBIBTEX produces the same outputs as BIBTEX from an 'aesthetic' point of view. Readers interested in this topic can consult some good typography manuals: for example, [10, § 10], [11, § 15.54–15.76], [22, p. 53–54]. French-speaking readers can also refer to [12, § 94] or [34, p. 31–36] about the conventions ruling French bibliographies, German-speaking readers can refer to [15], too. More official documents have been issued by the ANSI[4] [3], or *British Standards* [8, 9], or the French institute of standardisation[5] [1].

We assume that readers are familiar with the usual commands of LATEX and BIBTEX[6] and the way to use them in cooperation. However, we recall some points in order to make precise our terminology. A BIBTEX file (with the .bib suffix) contains a series of **entries** like that given in Figure 1. When BIBTEX runs, it builds a .bbl file that contains **references** for a LATEX document, using the `\bibitem` command, and according to a *bibliography style*. For example, the reference corresponding to Entry `gibson1988` (cf. Figure 1) and using the plain bibliography style—that is, references are labelled with numbers—will look like:

[1] William Gibson. *Mona Lisa Overdrive*. Victor Gollancz, Ltd., 1988.

after being processed by LATEX. There exist many other bibliography styles: most of them are described in [21, § 13.2], more in [25, § 4.3]—including some styles suitable for French bibliographies—and keys to design new bibliography styles are given in [37].

The sections of this article will successively explore first the multilingual exten-

[3] See `http://lifc.univ-fcomte.fr/PEOPLE/hufflen/texts/mlbibtex/mlbibtex/mlbibtex.html`.
[4] **A**merican **N**ational **S**tandards **I**nstitute.
[5] *AFNOR: **A**ssociation **F**rançaise de **NOR**malisation.*
[6] The basic reference is [32]. There are also [14] and [43] in French, [29, 30, 31] in German.

sions provided and processed by MLBIBTEX, then the ways of compatibility between BIBTEX's current implementation and ours. Finally we briefly describe our implementation and we conclude with what we plan for the future of MLBIBTEX. An annex sums up all our syntactic conventions.

The Extensions of mlbibtex

In this section, we present the extensions provided by MLBIBTEX. We attempt to detail them in an informal but precise way. Besides, we show that they meet actual requirements about multilingual bibliographies.

MLBIBTEX allows its users to specify *language changes* and *language switches*. Before we describe them, let us explain what 'the bibliographical entry's language' and 'the bibliographical reference's language' mean. In fact, there are two approaches for multilingual bibliographies.

- ◇ According to the first approach, the information related to a reference should be expressed using the language of the corresponding document. For example, the month of issue should be 'July' for a reference about a document written in English, '*juillet*' for a reference about a document written in French, '*Juli*' for a reference about a document written in German. Roughly speaking, the values to be put into the fields of BIBTEX entries are copied slavishly from what is printed on the document. From a 'philosophical' point of view, this convention proceeds from the idea that a reference is wholly suitable only for people reading the language this referred document is written in[7]. We have to add language information to each entry, in the sense that LATEX must be able to format the corresponding reference by using the typographical conventions of this language. Such language information is given by an additional MLBIBTEX field called `LANGUAGE`[8]. The value of this field defaults to `english`. In the following, this approach will be called **reference-dependent**.
- ◇ Given a printed work, the second approach consists of using its language for the information of all its bibliographical references, as far as possible. So, in comparison with the example illustrating the first approach, all the months of issue should be expressed in English if the work is written in English. In the same way, they should be expressed in French (resp. German) if the work is written in French (resp. German). However, if this approach is systematic, some information fields other than dates should be superseded by a translated form. For example, let us consider the entry given in Figure 2, concerning the novelisation of a film. This reference can be formatted as it is within a bibliography in English, but the `NOTE`

[7] Following this approach to extremes, the information related to a document written in a language using a non-Latin alphabet should be put using this alphabet's characters. For example, 'июль' for 'July' in Russian. But we can remark that if the title of such a document is transliterated into Roman letters, this operation needs to know the original language when the bibliographical entry is established.

[8] In fact, this 'new' field has already been used in conjunction with the mlbib package [35]. BIBTEX—like MLBIBTEX—allows end-users to add new fields, which are ignored by 'standard' bibliography styles.

```
@BOOK{bisson1995,
        AUTHOR = {Terry Bisson},
        TITLE = {{J}ohnny {M}nemonic},
        NOTE = {Based on a short story and screenplay by William Gibson},
        PUBLISHER = {Simon \&~Shuster, Inc.},
        YEAR = 1995}
```

Figure 2: A BIBTEX entry, but suitable for a bibliography in English.

field should be replaced by:

D'après une histoire et un scénario de William Gibson

within a bibliography in French. In the following, this approach will be called **document-dependent**.

From our point of view, the choice between these two approaches does not have to be made by the designer of a bibliography program like BIBTEX. This choice proceeds from personal considerations, or requirements from an editor or a publisher. That is why we think that a multilingual bibliography program associated with LATEX should be able to put both these two approaches into action. To emphasise this point, let us consider a book like an anthology or proceedings, with several chapters written in several languages. If each chapter ends with its own bibliography and if the publisher requires that the bibliography of a chapter must be written in the chapter's language, a reference—for example, a reference to Entry `bisson1995` above—cited in two chapters written in English and French will appear differently within the two bibliographies. In this case, we show that it should be possible for a bibliography program to be adapted to several languages used within a document, as well as it should be possible for an entry to be used to generate different references according to the language chosen.

There exist some LATEX2ϵ packages for each of these two approaches. About the reference-dependent approach, the mlbib package [35] uses a `LANGUAGE` field and allows each item of a bibiliography to be printed according to suitable typography. About the document-dependent approach, the oxford package [4] allows users to choose a language for the whole of the bibliography. But these implementations are partial: only basic typographical rules (rules for spacing in French, for example) and some keywords (month names, for example) are taken into account. The same fact holds about bibliography styles built with the makebst program (cf. [13] or [21, § 13.9]) and interfaced with the babel package. If 'standard' BIBTEX is used, there is no 'actual' multilinguism in entries, unless users fill in some fields by using commands originating from the multilingual packages of LATEX2ϵ.

So, here are the precise definitions related to our terminology.

◇ A **language identifier** is a non-ambiguous prefix of:
- an option of the babel package,
- or a multilingual package name such as french or german[9].

[9]This choice of a non-ambiguous prefix allows a language identifier to get access to several ways to process a language. For example, a language identifier set to `french` works with the frenchb option [18]

```
@BOOK{king1982f,
        AUTHOR = {Stephen~Edwin King},
        TITLE = {The Running Man},
        NOTE = {[Written as] * english
                [Sous le pseudonyme de] * french
                [Unter der Pseudonym] * german
                Richard Bachman},
        PUBLISHER = {New American Library},
        YEAR = 1982,
        MONTH = may,
        LANGUAGE = english}
```

Figure 3: Example of a multilingual entry.

- ◇ The **entry's language** is given by the `LANGUAGE` field[10].
- ◇ The **reference's language** is given:
  - – either by the `LANGUAGE` field if each item of the bibliography should be expressed in its own language (reference-dependent approach),
  - – or by the language in which the document is written if this language is to be used for the whole of the bibliography (document-dependent approach).

As mentioned above, this convention about the reference's language allows us to put both possible approaches for multilingual bibliographies into action.

Language switches

There are two kinds of language switches, with and without a default language. They are used for information about what must be put, possibly in another language, and for details that can be given in a particular language, but can be omitted if no translation is available.

SWITCH WITH A DEFAULT LANGUAGE It is expressed by the following syntax:

$$\texttt{[}\mathit{string}_0\texttt{]}\ \texttt{*}\ \mathit{idf}_0\ \texttt{[}\mathit{string}_1\texttt{]}\ \texttt{*}\ \mathit{idf}_1\ \ldots\ \texttt{[}\mathit{string}_n\texttt{]}\ \texttt{*}\ \mathit{idf}_n \qquad (1)$$

where string_0, string_1, …, string_n ($n \in \mathbb{N}$) are strings of characters[11] and idf_0, idf_1, …, idf_n are pairwise-different language identifiers. MLBIBTEX processes it as follows:

- ◇ if there exists i ($0 \leq i \leq n$) such that the reference's language is equal to idf_i, Expression (1) yields string_i;
- ◇ otherwise, MLBIBTEX puts the string associated with the value of the `LANGUAGE` field: if such a value does not exist, Expression (1) is replaced by an empty string.

of the babel package as well as the french package [19]. Readers who are interested in a comparative study between these two ways to write French documents can consult [23, § 2].

[10] Let us recall that it defaults to the `english` value.

[11] We follow the convention originating from BIBTEX, that is, a group surrounded by braces ('`{...}`') is viewed as a single character. We will go thoroughly into this point in Subsection *Using square brackets as syntactic tokens*.

```
@BOOK{king1990,
        AUTHOR = {Stephen~Edwin King},
        TITLE = {The Stand},
        NOTE = {[The Complete and Uncut Edition.] * english
                [Version int\'{e}grale.] * french
                [Abridged version issued in 1978] * english
                [Version abr\'{e}g\'{e}e parue en 1978] * french
                [Abgek\"{u}rzt Auffassung im Jahre 1978 erschienen] * german},
        PUBLISHER = {Doubleday \&~C\textsuperscript{o}},
        ADDRESS = {New-York},
        YEAR = 1990,
        LANGUAGE = english}
```

Figure 4: Example with two language switches.

An example is given in Figure 3. When using the plain bibliography style, MLBIBTEX will produce the following references, after processing by LATEX:

◇ when the reference's language is English:

[1] Stephen Edwin King. *The Running Man*. New American Library, May 1982. Written as Richard Bachman.

◇ when it is French:

[1] Stephen Edwin King. *The Running Man*. New American Library, mai 1982. Sous le pseudonyme de Richard Bachman.

◇ when it is German:

[1] Stephen Edwin King. *The Running Man*. New American Library, Mai 1982. Unter der Pseudonym Richard Bachman.

As an example of using default values, if MLBIBTEX is asked for a Russian-speaking reference, the information in English is put since we did not specify any string for the Russian language, and the value of the `LANGUAGE` field is `english`. In fact, only the month name—given by the abbreviation `may` that BIBTEX and MLBIBTEX know—is printed in Russian:

[1] Stephen Edwin King. *The Running Man*. New American Library, май 1982. Writing as Richard Bachman.

'`* idf`' where *idf* is the value of the `LANGUAGE` field can be omitted. The example given in Figure 3 about the `NOTE` field could be abridged as follows:

```
NOTE = {[Written as]
        [Sous le pseudonyme de] * french
        [Unter der Pseudonym] * german
        Richard Bachman}
```

A language switch ends:

◇ either at the end of a MLBIBTEX field or before a common part, not surrounded by square brackets: this second convention holds for the example above, the common part being '`Richard Bachman`';

```
@BOOK{king1978f,
        AUTHOR = {Stephen~Edwin King},
        TITLE = {Night Shift},
        NOTE = {[Collection of 20 short stories] * english
                [Recueil de 20~nouvelles] * french
                [Titre de la traduction fran\c{c}aise : Danse macabre] * french
                [Titel der deutschen \"{U}bersetzung: Nachtschicht] * german},
        PUBLISHER = {Doubleday \&~C\textsuperscript{o}},
        YEAR = 1978.
        LANGUAGE = english}
```

Figure 5: Two language switches, too.

◇ when a language identifier is repeated, in which case another language switch begins. For example, look at Figure 4. There are two switches, the first specifying a choice between English and French, the second a more complete choice among English, French and German, the default language being English in both cases. If this entry is used to produce a German-speaking reference, that will result in using the default value for the first switch and an *ad hoc* value for the second:

> [1] Stephen Edwin King. *The Stand*. Doubleday & C°, New-York, 1990. The Complete and Uncut Edition. Abgekürzt Auffassung im Jahre 1978 erschienen.

In order to make language switches more readable, we recommend end-users to begin each of them with the default value. For example, look at Figure 5: the `NOTE` field contains two language switches: the first specifying a choice between English and French, defaulting to English, the second a choice between French and German, without default. Even if MLBIBTEX is able to process the `NOTE` field from Figure 5 without any trouble, we think that it should be put down using this more readable form:

```
NOTE = {[Collection of 20~short stories]
        [Recueil de 20~nouvelles] * french
        []
        [Titre de la traduction fran\c{c}aise : Danse macabre] * french
        [Titel der deutschen \"{U}bersetzung: Nachtschicht] * german}
```

A language switch with a default language may occur anywhere except:

◇ within a `LANGUAGE` field, which *must* be a language identifier if defined,

◇ or within `CROSSREF` or `KEY` fields.

In MLBIBTEX's Version 1.1, a language switch with a default language cannot occur within `AUTHOR` or `EDITOR` fields, either. We plan to implement this last feature in Version 1.2, because it can be useful for the transliteration of person names originating from a language using a non-Latin alphabet. Since this transliteration is generally phonetic, it may be different from one language to another, as shown by the following example:

```
AUTHOR = {[Арам Ильич Хачатурян] * russian
```

```
@BOOK{gibson1986,
        AUTHOR = {William Gibson},
        TITLE = {Burning Chrome and Other Stories},
        NOTE = {[Titre de la traduction fran\c{c}aise :
                 \emph{Grav\'{e} sur chrome}] ! french
                [Titel der deutschen \"{U}bersetzung: \emph{Cyberspace}] ! german},
        PUBLISHER = {Victor Gollancz, Ltd.},
        YEAR = 1986,
        LANGUAGE = english}
```

Figure 6: Example of language switch without default language.

```
          [Aram Il'yich Khachaturian] * english
          [Aram Ilyich Khatchatourian] * french
          [Aram Iljitsch Chatschaturjan] * german},
...
LANGUAGE = ...
```

It is for this reason—related to the possible use of other alphabets—that we allow language switches within fields such as:

ADDRESS	JOURNAL	SCHOOL
BOOKTITLE	ORGANIZATION	SERIES
INSTITUTION	PUBLISHER	TITLE

In the same way, the date may be expressed using different calendars, which is why MLBIBTEX's future versions will probably allow language switches with a default language within fields such as MONTH or YEAR.

SWITCH WITHOUT DEFAULT LANGUAGE It is expressed by the following syntax:

$$\texttt{[}\mathit{string}_0\texttt{] ! }\mathit{idf}_0\texttt{ [}\mathit{string}_1\texttt{] ! }\mathit{idf}_1\texttt{ ... [}\mathit{string}_n\texttt{] ! }\mathit{idf}_n \qquad (2)$$

where n, string_0, string_1, ..., string_n, idf_0, idf_1, ..., idf_n have the same meaning as in Expression (1). MLBIBTEX processes it as follows:

- ◇ it behaves exactly like a language switch with '*' if there exists i ($0 \leq i \leq n$) such that the reference's language is equal to idf_i, that is, Expression (2) yields string_i;
- ◇ otherwise, Expression (2) is replaced by an empty string, and a warning message is emitted by MLBIBTEX.

An example is given in Figure 6. When using the plain bibliography style, MLBIBTEX will produce the following references, after processing by LATEX:

- ◇ when the reference's language is French:

  [1] William Gibson. *Burning Chrome and Other Stories*. Victor Gollancz, Ltd., 1986. Titre de la traduction française : *Gravé sur chrome*.

- ◇ when it is German:

```
@BOOK{king1981i,
        AUTHOR = {Stephen~Edwin King},
        TITLE = {[Danse macabre] : french},
        PUBLISHER = {Everest House},
        YEAR = 1981,
        MONTH = jul,
        LANGUAGE = english}
```

Figure 7: Example of a language change.

[1] William Gibson. *Burning Chrome and Other Stories*. Victor Gollancz, Ltd., 1986. Titel der deutschen Übersetzung: *Cyberspace*.

◇ and an empty note will be put otherwise:

[1] William Gibson. *Burning Chrome and Other Stories*. Victor Gollancz, Ltd., 1986.

A language switch without default language may occur anywhere except within following fields:

AUTHOR	JOURNAL	TITLE
BOOKTITLE	KEY	YEAR
CROSSREF	LANGUAGE	
EDITOR	MONTH	

Language change
It is expressed by the following syntax:

$$\texttt{[}\textit{string}\texttt{]} : \textit{idf} \tag{3}$$

where *string* is a string of characters, and *idf* a language identifier. It expresses conformity to other typographical conventions and can be used to hyphenate foreign words: here 'foreign' means 'belonging to a language different from the value of the LANGUAGE field'. For example, the language change given within the TITLE field in Figure 7 ensures that the title, using French words for an American book, will be hyphenated correctly if need be.

A language change may occur anywhere except within the following fields:

CROSSREF	LANGUAGE	YEAR
KEY	MONTH	

In MLBIBTEX's Version 1.1, a language change cannot occur within AUTHOR or EDITOR fields either, but we plan to implement this feature in Version 1.2.

Using square brackets as syntactic tokens
In MLBIBTEX, the use of square brackets does not interfere with the different meanings of braces. It is known that in BIBTEX (cf. [21, § 13.5.2]), the two following field specifications:

```
{The Eyes of the Dragon}     "The Eyes of the Dragon"
```
(4)

are equivalent. Using braces to surround the whole of a field—like in the expression at the left—is needed when this field contains the double-quote character:

```
{"For the Love of Barbara Allen"}
```

In fact, a double-quote character, as part as a field's value, must be surrounded by braces. The following value:

```
"{"For the Love of Barbara Allen"}"
```

is correct. Anyway, these two field specifications are equivalent, too:

```
{{The Eyes of the Dragon}}   "{The Eyes of the Dragon}"          (5)
```

but Expressions (4) make possible for this title to be non-capitalised, whereas Expressions (5) tells BIBTEX to consider the string `The Eyes of the Dragon` as it is[12]. In the same way, the following specifications are equivalent in MLBIBTEX:

```
{[Firestarter] [Charlie] * french [Feuerkind] * german}
"[Firestarter] [Charlie] * french [Feuerkind] * german"          (6)
```

and so are the following four:

```
{{[Firestarter] [Charlie] * french [Feuerkind] * german}}
{[{Firestarter}] [{Charlie}] * french [{Feuerkind}] * german}
"{[Firestarter] [Charlie] * french [Feuerkind] * german}"
"[{Firestarter}] [{Charlie}] * french [{Feuerkind}] * german"    (7)
```

but Expressions (6)—resp. (7)—are analogous to Expressions (4)—resp. (5)—with respect to possible capitalisation.

Square brackets must be balanced, unless they are put in math mode—that is, enclosed between two '`$`' characters—in which case they do not have any meaning for MLBIBTEX. A double-quote character belonging to a field's value must be surrounded by braces or square brackets. If square brackets are used like in this example:

```
TITLE = {The Girl Who [Loved] Tom Gordon},
```

MLBIBTEX believes that they surround the default part of a language switch with '`*`', the '`*`' character followed by the default language being omitted. So the result will be:

The Girl Who Loved Tom Gordon

in any case.

If users wish to insert them within the value of a field[13], the right square bracket must be followed by the empty string:

```
NOTE = {Brilliant\ldots\ A delight to read [\ldots] {} A true original.
        (\emph{Sunday Times})}
```

[12]This rule does not hold if the enclosed opening brace is followed by a TEX or LATEX command. For example, '`{{\relax The Eyes of the Dragon}}`' produces the same outputs as one of Expressions (4): cf. [21, § 15.5.2] and [28, Ch. 24] about the `\relax` command. This way, MLBIBTEX behaves exactly like BIBTEX.

[13]Let us recall that in typography, square brackets are used to enclose editorial interpretations, corrections, explanations, ... cf. [11, § 5.128–5.132]. '[...]' means that some words have been skipped.

The same convention holds if users wish to print square brackets followed by '`*`', '`!`' or '`:`' characters:

```
{Read [Trails in Darkness] {} * by Robert~Erwin Howard}
```

From our point of view, the two drawbacks of its convention are:

◇ if users want to insert a left square bracket only, or a right square bracket only, the solution is to use the LaTeX command `\symbol`:

```
{\symbol{91}Count Zero}        for '['
{Virtual Light\symbol{93}}     for ']'
```

◇ the same 'trick' holds when users wish square brackets to be nested, which is not allowed by MlBibTeX:

```
{Read [\symbol{91}Eons of the Night\symbol{93}] {} by Robert~Erwin Howard}
```

How the language is determined

MlBibTeX considers a **bibliography's language**. Version 1.1 can process documents written:

◇ with the default configuration of LaTeX, that is, without any multilingual package, in which case, the bibliography's language is supposed to be `english`;

◇ by using the babel package, in which case the bibliography's language is the default language, that is, the last option when this package is loaded[14].

Concerning other mulilingual packages, the bibliography's language will be `french` or `german` when MlBibTeX works with the french [19] and german [41] packages, which is planned for Version 1.2.

If the bibliography style used is reference-dependent, the reference's language is given by the `LANGUAGE` field for each entry. If it is document-dependent, the reference's language is always the bibliography's language, but MlBibTeX considers the value given by the `LANGUAGE` field for the following information fields:

```
AUTHOR       EDITOR     PUBLISHER
BOOKTITLE    JOURNAL    TITLE
```

which are processed according to suitable typographical conventions when there is no language switch. To explain this behaviour, let us consider Entry `king1998d` in Figure 8, and let us assume that we would like to refer it in a document written in French. If the bibliography style is reference-dependent, the corresponding reference will be wholly written in English and the result, in the .bbl file, will look like—cf. [7, § 12.2] about the `otherlanguage` environment—:

```
\bibitem{king1998d}
\begin{otherlanguage}{english}Stephen~Edwin King. \emph{Bag of Bones}.
Scribner Book Company, September 1998. Translated into French and German.
\end{otherlanguage}
```

If the bibliography style is document-dependent, only the fields supposed to be in English will be processed according to English typographical conventions, that is, the `TITLE` field, but not the `PUBLISHER` field because there is a language switch with no

[14] This default language is given by the `\bbl@main@language` command, provided by the babel package: cf. [7, § 12.2].

```
@BOOK{king1998d,
        AUTHOR = {Stephen~Edwin King},
        TITLE = {Bag of Bones},
        NOTE = {[Translated into French and German] * english
                [Titre de la traduction fran\c{c}aise : Sac d'os] * french
                [Titel der deutschen \"{U}bersetzung: Sara] * german},
        PUBLISHER = {[Scribner Book Company] * english
                     [Premi\'{e}re \'{e}dition am\'{e}ricaine] * french},
        YEAR = 1998,
        MONTH = sep,
        LANGUAGE = english}
```

Figure 8: Yet another example of language switch.

common part. (In addition, let us recall that MLBIBTEX's Version 1.1 processes the `AUTHOR` field exactly like BIBTEX, that is, it does not switch over to another language for this field.) So, the result in the .bbl file will look like:—cf. [7, § 12.2] about the `\foreignlanguage` command—:

```
\bibitem{king1998d}
Stephen~Edwin King. \foreignlanguage{english}{\emph{Bag of Bones}}.
Premi\'{e}re \'{e}dition am\'{e}ricaine, septembre 1998. Titre de la
traduction fran\c{c}aise : Sac d'os.
```

If some fields $F_0, \ldots, F_n$ ($n \geq 0$) are provided by an entry E accessed by means of a `CROSSREF` field, these fields $F_0, \ldots, F_n$ are processed by considering the value of the `LANGUAGE` field for E.

Whenever it has to switch over to another language, MLBIBTEX checks that this language will be known when LATEX processes the .bbl file. If not, this other language is replaced by the bibliography's language[15]. This behaviour is different from what happens if direct commands originating from the babel package are used within the value of a BIBTEX field. For example:

```
{\iflanguage{frenchb}{Jessie}{\iflanguage{german}{Das Spiel}{Gerald's Game}}}
```

—cf. [7, § 12.2] about the `\iflanguage` command—will work only if the corresponding reference belongs to the bibliography of a document using the babel package with at least the frenchb and german options, which can be an actual drawback if .bib files are shared out among several people.

As an example, the bibliography of this article has been put with a reference-dependent bibliography style.

Issue of Compatibility

Due to the huge number of .bib files already written, MLBIBTEX *should* be able to work with them. We ought to have written 'has to' instead of 'should', but syntactic

[15]So, for the example above, we assumed that the babel package was loaded in the LATEX document, with at least the english and frenchb (or french) options.

◇ Commands:

month names	ordinal numbers	other keywords
\bbljan	\bblfirsto	\bbland
\bblfeb	\bblsecondo	\bblchap
\bblmar	\bblthirdo	\bbled
\bblapr	\bblfourtho	\bbledby
\bblmay	\bblfiftho	\bbledn
\bbljun	\bblst	\bbleds
\bbljul	\bblnd	\bblin
\bblaug	\bblrd	\bblmasterthesis
\bblsep	\bblth	\bblno
\bbloct		\bblof
\bblnov		\bblp
\bbldec		\bblphdthesis
		\bblpp
		\bbltechrep
		\bblvol

The commands for month names implement the abbreviations known by BIBTEX and MLBIBTEX:

```
jan   feb   mar   apr   may   jun   jul   aug   sep   oct   nov   dec
```

◇ The `bblquotedtitle` allows a title to be quoted:

```
\begin{bblquotedtitle}%
Nesnesiteln\'{a} lehkost byt\'{\i}
\end{bblquotedtitle}
```

– with respect to English style: “The Unbearable Lightness of Being”
– w.r.t. French style: « L'insoutenable légèreté de l'être »
– w.r.t. German style: „Die Unerträgliche Leichtigkeit des Seins“
– and other styles in interface with MLBIBTEX.

By the way, notice the use of the “`%`” character—which opens a comment—in the example above, just after “`\begin{bblquotedtitle}`”. This aims to bypass the end-of-line character, because LATEX considers it as a space character (*cf.* [32, § 2.2.1]). Another way to avoid any undesirable spacing is to glue the opening of this environment “`\begin{bblquotedtitle}`” and the beginning of the quoted section together, without any space character:

```
\begin{bblquotedtitle}Neuromancer
\end{bblquotedtitle}
```

Table 1: Additional LATEX commands used in bibliographies generated by MLBIBTEX.

conventions have changed... In the same way, some end-users of BIBTEX developed their own bibliography styles, and these styles *should* be able to be used with our version. ‘Should be able to be used...’, too. Here is what we describe precisely below.

Roughly speaking, the whole distribution of MLBIBTEX consists of:

◇ an executable program, called ‘MLBIBTEX’;

◇ a file defining some additional LATEX commands[16] given in Table 1;

[16] Most of them are used in bibliography styles generated by the makebst program, and in interface with the babel package (cf. [13] or [21, § 13.9]). In this case, they have to be defined in a file called babelbst.tex.

◇ some bibliography style files that can be used in addition to the standard bibliography style files. Some begin with the command:

```
REFERENCEDEPENDENT
```

so they put the reference-dependent approach into action. Otherwise, they are document-dependent.

Any existing bibliography style file should work with MLBIBTEX, provided it works with BIBTEX's current implementation[17]. Due to our conventions, these existing bibliography style files are supposed to be document-dependent.

As far as we know, there are only two bug cases when BIBTEX (.bib) files are used with MLBIBTEX:

◇ unbalanced square brackets,

◇ nested square brackets,

—see Subsection *Using square brackets as syntactic tokens*—but we think that they must be very rare in real situations. Anyway, MLBIBTEX knows any field name BIBTEX knows, including names put in compatibility with SCRIBE [42]. If a 'new-fashioned' style is used for files processed by current BIBTEX, there is a bug case:

◇ double-quote character surrounded by square brackets, but not by braces, for example:

```
"Virtual ["Light"]"
```

in the same way, that should hardly ever happen. Otherwise, these files should be processed without any bug but obviously the result may look somewhat strange.

The additional bibliography style files can be used with BIBTEX's current implementation, provided that the `REFERENCEDEPENDENT` command is removed and the LATEX commands given in Table 1 are defined when the .bbl file is processed. So they behave in a 'standard' way. Besides, MLBIBTEX uses two environment variables:

`MLBIBINPUTS` when it searches for a .bib file,
`MLBSTINPUTS` ..... .bst file,

the corresponding environment variables used by BIBTEX's current implementation being `BIBINPUTS` and `BSTINPUTS`. These two sets of environment variables can be given suitable values, so end-users can ensure that 'new-fashioned' files are processed only by MLBIBTEX.

Some Words about the Implementation

In order to be able to master a program in constant progress, we chose to develop MLBIBTEX from scratch, even if we confess that we often consulted the source files of current BIBTEX to get as much experience as possible.

[17]Bug reports or improvement suggestions will be welcome at hufflen@lifc.univ-fcomte.fr. In addition, we will progressively put all the bug reports and any additional information onto the Web page `http://lifc.univ-fcomte.fr/PEOPLE/hufflen/texts/mlbibtex/mlbibtex/mlbibtex.html`.

MLBIBTEX is written in the C programming language [26], since it has become standard and is efficient and widely available on many systems. Besides, its use allowed us to get access to many development tools, especially GNU[18] tools. For example, the scanner and parser have been developed using the GNU scanner and parser generators: flex and bison, corresponding to lex and yacc within 'standard' UNIX [33].

In fact, we adopted an approach of *reverse engineering*, since we recovered design form analysing source files and documentation. But this study allowed us to put a precise modular decomposition into action. This decomposition yielded a precise terminology to name functions and variables, in order to ease the possible improvement of some modules.

Conclusion

Developing the first version of MLBIBTEX was a real challenge for us, since we personally missed this kind of multilingual tool quite often. We also have been very interested in the problems raised by extending BIBTEX's grammar[19]. Concerning ergonomics, we think that our proposals are user-friendly and will need as few adaptations as possible when end-users put BIBTEX files according to MLBIBTEX conventions. But we do not have actual feedback, that is why we presently consider our program as a prototype, being about to belong to the LATEX *legacy* [44], and independent of BIBTEX's future version described in [39]. Our goal is to be fully able to perform some experiments... and other experiments. For this reason, we have preferred a step-by-step approach. So we could change our syntactic conventions if it appears to be preferable.

Besides, we did not forget that many tools are based on BIBTEX[20], and we think that adaptations should be slight if developers of such tools would like to make them conformant to MLBIBTEX, if need be.

On another subject, many extensions of MLBIBTEX are planned:

- ◇ a version based on Unicode,
- ◇ an extension allowing users to define sorting programmes with respect to the lexicographical order associated with a particular language[21],
- ◇ the extension of the language used in bibliography style files, in order to ease file inclusions and avoid the duplication of identical parts from one bibliography style to another.

[18] Recursive acronym for '**G**NU's **N**ot **U**nix'. This project aims to develop *free software*. For more details, see the Web page `http://www.gnu.org`.

[19] In fact, the use of square brackets as syntactic elements originates from our study of the CAMEL citator [6, 5].

[20] For example, bib2bib and bibtex2html, described in [17]. A list of tools based on BIBTEX can be found at the Web page `http://www.ecst.csuchico.edu/~jacobsd/bib/formats/bibtex.html`, maintained by Danna Jacobsen.

[21] Let us consider the Spanish language as an example: words beginning with '`ll...`' are alphabeticised after other words beginning with '`l...`', and before words beginning with '`m...`', that is:

`la...` `ly...` `lla...` `m...`

```
<the_axiom>             ::=   <information_list> ;
<information_list>      ::=   /* empty */ | <information> <information_list> ;
<information>           ::=   <comment_information> |
                              <preamble_information> |
                              <string_information> |
                              <entry_information> ;
<comment_information>   ::=   "@COMMENT" bibtex_comment ;
<preamble_information>  ::=   "@PREAMBLE" <bibtex_value> ;
<string_information>    ::=
    "@STRING" ("{" bibtex_identifier "=" <bibtex_expression> "}" |
               "(" bibtex_identifier "=" <bibtex_expression> ")") ;
<entry_information>   ::=
    entry_keyword ("{" label <after_label> "}" | "(" label <after_label> ")") ;
<after_label>         ::=   /* empty */ | "," <field_list> ;
<field_list:          ::=   /* empty */ | <field_nelist> ;
<field_nelist>        ::=   <field> | <field> "," <field_nelist> ;
<field>               ::=   <field_name> "=" <bibtex_expression> ;
<bibtex_expression>   ::=   <concatenation_list> ;
<concatenation_list>  ::=   <bibtex_value> | <bibtex_value> "#" <concatenation_list> ;
<bibtex_value>        ::=   bibtex_identifier | natural_number | mlbibtex_atomic_value ;
```

Table 2: Grammar of BIBTEX (and MLBIBTEX).

And finally, we also plan a comparative study of the multilingual features of MLBIBTEX and those provided by XML and XSL[22], which could help us develop further versions of MLBIBTEX.

Annex: mlbibtex's Syntax

Now we outline our syntax precisely. We plan to update this description at each syntax change, in order to ease the writing of tools associated with MLBIBTEX.

MLBIBTEX's parser—which could be used as BIBTEX's parser—is based on the grammar given in Table 2. This grammar is expressed with a formalism close to BNF[23], that is:

- ◇ for each nonterminal symbol, enclosed by '<' and '>', the expression following the '::=' sign and terminated by ';' states how it can be expanded: for example, the non-terminal symbol <the_axiom> can be expanded only to the non-terminal symbol <information_list>;
- ◇ if a non-terminal symbol expands to an empty expression, we emphasise that by putting a C-like comment /* empty */;
- ◇ the nonterminal symbol from which the whole grammar is derived, is <the_axiom>;

[22] 'e**X**tensible **M**arkup **L**anguage' and 'e**X**tensible **S**tyle **L**anguage'. A good introduction to these languages is [20], in French, and an updated version is available in English: see the Web page http://webcast.cern.ch/Projects/WebUniversity/AcademicTraining/Goossens/.

[23] **B**ackus-**N**aur **F**orm. Readers unfamiliar with this formalism can refer to [33] for more details. A more general reference about these techniques is [2].

`bibtex_comment` All the characters are skipped, until next '`@`' character.

`bibtex_identifier` A *constituent* (see below), followed by zero or more constituents or digits.

`constituent` A letter (alphabetical character) or one of the following characters:

```
' ~ @ # $ % ^ & + * - / _ . : ; ? ! < > [ ] \
```

`entry_keyword` Syntactically, the '`@`' character, followed by zero or more constituents or digits, but `@COMMENT`, `@PREAMBLE` and `@STRING` are not entry keywords. Here are the entry keywords used most often:

`@ARTICLE`	`@CONFERENCE`	`@MANUAL`	`@PHDTHESIS`
`@BOOK`	`@INBOOK`	`@MASTERSTHESIS`	`@PROCEEDINGS`
`@BOOKLET`	`@INCOLLECTION`	`@MISC`	`@TECHREPORT`
`@COLLECTION`*	`@INPROCEEDINGS`	`@PATENT`*	`@UNPUBLISHED`

The entry keywords asterisked are not standard, they might occur within some `.bib` files.

`field_name` Syntactically, a letter, followed by zero or more constituents or digits. Here are the fields used more often:

`ABSTRACT`*	`EDITION`	`LCCN`*	`SCHOOL`
`ADDRESS`	`EDITOR`	`LOCATION`*	`SERIES`
`AFFILIATION`*	`HOWPUBLISHED`	`MONTH`	`SIZE`*
`ANNOTE`	`INSTITUTION`	`MRNUMBER`*	`TITLE`
`AUTHOR`	`ISBN`*	`NOTE`	`TYPE`
`BOOKTITLE`	`ISSN`*	`NUMBER`	`URL`*
`CHAPTER`	`JOURNAL`	`ORGANIZATION`	`VOLUME`
`CONTENTS`*	`KEY`	`PAGES`	`YEAR`
`COPYRIGHT`*	`KEYWORDS`*	`PRICE`*	
`CROSSREF`	`LANGUAGE`*	`PUBLISHER`	

The field names asterisked are not used in standard bibliography styles, but can be used by some additional styles or in some LaTeX2ϵ packages.

`label` A non empty-sequence of constituents or digits, the '`{`' and '`}`' characters being also allowed.

`natural_number` A non-empty sequence of digits. Negative numbers or numbers with '`.`' are not allowed.

Table 3: Lexical tokens of MLBibTeX, except for field's values.

◇ expressions enclosed by two double quote characters are reserved words and symbols,

◇ symbols not surrounded by braces are terminal, that is, they are tokens read by the scanner,

◇ the '`|`' sign means an alternative,

◇ parentheses are used to override precedence.

We describe the tokens common to BibTeX and MLBibTeX in Table 3, whereas the description of the grammar ruling the possible values for MLBibTeX's fields are given in Table 4.

Let us recall that:

◇ the reserved words of BibTeX and MLBibTeX are read by a case insensitive scan-

```
<mlbibtex_atomic_value>     ::=   '"' <no_double_quote>* '"' | '{' <between_braces> '}' ;
<no_double_quote>           ::=
    <almost_any> |
    '$' <any_math>* '$' |
    '[' <between_square_brackets>* ']' <after_square_brackets> |
    '{' <between_braces> '}' ;
<between_square_brackets>   ::=
    <almost_any> | '"' | '{' <between_square_brackets> '}' | '$' <any>* '$' ;
<after_square_brackets>   ::=
    /* empty */ |
    '*' <bibtex_identifier> |
    '!' <bibtex_identifier> |
    ':' <bibtex_identifier> |
    '{' '}' ;
<between_braces>   ::=   <no_double_quote> | '"' ;
```

where:

`<almost_any>` is for any characters—including the space, tabulation, or end-of-line character—but braces, square brackets, '"' and '$';

`<any_math>` is for any character—including the space, tabulation, or end-of-line character—but '$';

`'...'` states a character that actually appears within the value;

`<...>*` means 'zero or more occurrences of `<...>`'.

Table 4: Lexical grammar of MLBIBTEX values.

ner[24]: for example, '`@BOOK`', '`@book`', '`@Book`', '`BoOk`' are suitable for the entry type `BOOK`;

- ◇ on the contrary, the values of BIBTEX and MLBIBTEX fields—that is, the possible values for the `bibtex_atomic_token` token—keep their own case;
- ◇ BIBTEX and MLBIBTEX ignore any text that is not inside an entry, so the `@COMMENT` command, used to put any texts outside commands, is not really necessary;
- ◇ from a syntactic point of view, some labels used to refer BIBTEX or MLBIBTEX entries, may be accepted by them, but be wrong arguments of the LATEX commands `\bibitem` and `\cite`;
- ◇ space, tabulation, and end-of-line characters between syntactic categories are irrelevant in the specification given in Table 2, but are relevant in Tables 3 and 4.

Acknowledgements

Thanks to Oren Patashnik for his valuable comments about this article, and Simon Pepping who proof-read it.

[24] On the contrary, LATEX is case sensitive.

References

[1] AFNOR Z44-045 : *Documentation — Références bibliographiques — Contenu, forme et structure*. Norme disponible auprès de l'AFNOR, voir `http://www.afnor.fr`. Décembre 1987.

[2] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: *Compilers, Principles, Techniques and Tools*. Addison-Wesley Publishing Company. 1986.

[3] ANSI/NISO Z39.71–1999: *Holding Statements for Bibliographic Items*. See `http://www.ansi.org`. 1999.

[4] Peter Antman: *Oxford Style Package*. Version 0.4. 1997. See CTAN: `biblio/bibtex/contrib/oxford/`.

[5] Frank G. Bennett, Jr: *The* Law *Module for the* Camel *Bibliography Engine*. July 1995. See `http://www.loria.fr/services/tex/english/bibdex.html`.

[6] Frank G. Bennett, Jr: *User's Guide to the* Camel *Citator*. July 1995. See `http://www.loria.fr/services/tex/english/bibdex.html`.

[7] Joannes Braams: *Babel, a Multilingual Package for Use with* LaTeX*'s Standard Document Classes*. Version 3.7. February 2001. See CTAN: `macros/latex/required/babel/babel.dvi`.

[8] British Standards 1629:1989: *Recommendation for References to Published Materials*. See `http://bsonline.techindex.co.uk`. 1989.

[9] British Standards 5605:1990: *Recommendations for Citing and Referencing Published Materials*. See `http://bsonline.techindex.co.uk`. 1990.

[10] Judith Butcher: *Copy-Editing. The Cambridge Handbook for Editors, Authors, Publishers*. 3rd edition. Cambridge University Press. 1992.

[11] *The Chicago Manual of Style*. The University of Chicago Press. The 14th edition of a manual of style revised and expanded. 1993.

[12] *Code typographique. Choix de règles à l'usage des professionnels du livre*. Fédération nationale du personnel d'encadrement des industries polygraphiques et de la communication, Paris. 17e édition. 1993.

[13] Patrick W. Daly: *Customizing Bibliographic Style Files*. Version 3.2. February 1999. Part of LaTeX' distribution.

[14] Bernard Desgraupes : LaTeX. *Apprentissage, guide et référence*. Vuibert Informatique, Paris. 2000.

[15] Duden: *Rechtschreibung der deutschen Sprache und der Fremdwörter*. 19. Auflage. Bibliographisches Institut, Mannheim. 1986.

[16] Michael J. Ferguson: "A Multilingual TeX". *TUGboat*, Vol. 6, no. 2, p. 57–58. July 1985.

[17] Jean-Christophe Filliâtre and Claude Marché: BibTeX*2*HTML*: A Translator of* BibTeX *bibliographies into* HTML. February 2001. See `http://www.loria.fr/services/tex/english/outils.html`.

[18] Daniel Flipo: *A babel Language Definition File for French*. Version v1.5e.

March 2001. See `http://www.tex.ac.uk`.

[19] Bernard GAULLE : *Notice d'utilisation du style french multilingue pour* LATEX. Version pro V5.01. Janvier 2001. Voir sur CTAN : `loria/language/french/pro/french/ALIRE.pdf`.

[20] Michel GOOSSENS : « XML et XSL : un nouveau départ pour le Web ». *Cahiers GUTenberg*, Vol. 33–34, p. 3–126. Novembre 1999.

[21] Michel GOOSSENS, Frank MITTELBACH and Alexander SAMARIN: *The* LATEX *Companion*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1994.

[22] *Hart's Rules for Composers and Readers at the University Press*. Oxford University Press. 39th edition. 1999.

[23] Jean-Michel HUFFLEN : « Typographie : les conventions, la tradition, les goûts, ... et LATEX ». *Cahiers GUTenberg*, Vol. 35–36, p. 169–214. Mai 2000.

[24] Jean-Michel HUFFLEN : « Vers une extension multilingue de BIBTEX ». *Cahiers GUTenberg*, Vol. 39–40, p. 23–38. Mai 2001.

[25] Jean-Michel HUFFLEN, Denis RŒGEL et Karl TOMBRE : *Guide local* (LA)TEX *du LORIA. Millésime 1998*. Rapport technique 98–R–214, LORIA. Septembre 1998.

[26] Brian W. KERNIGHAN and Denis M. RITCHIE: *The C Programming Language*. 2nd edition. Prentice Hall. 1988.

[27] Donal Ervin KNUTH and Pierre MACKAY: "Mixing Right-to-Left Texts with Left-to-Right Texts". *TUGboat*, Vol. 8, no. 1, p. 14–25. April 1987.

[28] Donald Ervin KNUTH: *Computers & Typesetting. Vol. A: the TEXbook*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1984.

[29] Helmut KOPKA: LATEX—*Band I: Einführung*. 2. Auflage. Addison-Wesley Longmann. 1997.

[30] Helmut KOPKA: LATEX—*Band II: Ergänzungen mit einer Einführung in* METAFONT. 2. Auflage. Addison-Wesley Longmann. 1997.

[31] Helmut KOPKA: LATEX—*Band III: Erweiterungen*. Addison-Wesley Longmann. 1997.

[32] Leslie LAMPORT: LATEX. *A Document Preparation System. User's Guide and Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1994.

[33] John LEVINE, Tony MASON and Doug BROWN: *lex & yacc*. 2nd edition. O'Reilly. October 1992.

[34] *Lexique des règles typographiques en usage à l'Imprimerie Nationale*. Imprimerie Nationale. 1990.

[35] Wenzel MATIASKE: *Multilinguale Zitierformate*. Oktober 1995. Siehe CTAN: `macros/latex/contrib/supported/mlbib/`.

[36] NTS TEAM and Peter BREITENLOHNER: *The ε-TEX Manual*. February 1998.

Part of LaTeX' distribution.

[37] Oren Patashnik: *Designing* BibTeX *styles*. February 1988. Part of LaTeX' distribution.

[38] Oren Patashnik: BibTeX*ing*. February 1988. Part of LaTeX' distribution.

[39] Oren Patashnik: "BibTeX 1.0". *TUGboat*, Vol. 15, no. 3, p. 269–273. September 1994.

[40] John Plaice and Yannis Haralambous: *Draft Documentation for the* Ω *System*. March 1998. See `http://www.loria.fr/services/tex/english/moteurs.html`.

[41] Bernd Raichle: *Die Makropakete „german" und „ngerman" für* LaTeX*2ε, LaTeX 2.09, Plain-TEX and andere darauf Basierende Formate*. Version 2.5. Juli 1998. Im Software LaTeX.

[42] Brian Reid: Scribe *Document Production System User Manual*. Technical Report, Unilogic, Ltd. 1984.

[43] Christian Rolland : LaTeX *par la pratique*. Éditions O'Reilly. Octobre 1999.

[44] Chris A. Rowley: "The LaTeX Legacy. 2.09 and All That". In: *Annual* ACM *Symposium on Principles of Distributed Computing*. Newport, Rhode Island. August 2001.

[45] *The Unicode Standard Version 3.0*. Addison-Wesley. February 2000.

♢ ♢ ♢

Special Fonts

Bogusław Jackowski* and Krzysztof Leszczyński†

abstract. We propose the use of a special pseudofont as an enhancement (in a sense) of the `\special` instruction. The examples of the implementation show that the technique applied here would prove to be extremely useful, especially with METAPOST.

keywords: cmdfont, special commands, MetaPost, fonts, PostScript

Keen users of TEX, METAFONT, or METAPOST might find the instructions called "special" very mighty helpers. However, METAPOST imposes serious limit on them: their content is placed at the very beginning of a PostScript file that METAPOST produces, just after the `%%Page` comment, before the very first real PostScript statement. It means that METAPOST, unlike TEX and METAFONT, is not able to intersperse drawing commands (`draw`, `fill`) or typesetting commands (`infont`, `btex ... etex`) with a special user-defined content. This behaviour embitters the life of METAPOST users and leads to neck-breaking solutions. Even worse, TEX `\special` instructions are ignored by dvitomp, making oodles of TEX packages unusable inside a `btex ... etex` construct.

The solution is to replace each `\special` instruction with a string typeset with a `special`-ly crafted font (pseudofont). We propose a natural name for it: `cmdfont`, *command font*. The text typeset with `cmdfont` has a special meaning when a TEX-generated dvi or a METAPOST-generated eps file is processed—it is treated as a sequence of commands to be interpreted.

The current article is the result of very preliminary thoughts—the idea is still very fresh. We are far from understanding all the consequences of this approach. Therefore, instead of developing a "general theory of specials," we decided to present just a few examples illustrating various possible applications of *special* `\special` *instructions*.

We have concentrated on using `cmdfont` with METAPOST. The use of pseudofonts with TEX, METAFONT, html or even major office editors is another thing. The reader may wish to evaluate the possibilities of special fonts. We perceive them as quite promising.

*B.Jackowski@GUST.org.pl
†Polish Linux Users' Group, chris@linux.org.pl

What is the special font?

Our special font can be defined with a short METAPOST program, `cmdfont.mp`:

```
designsize:=10bp/pt - epsilon;
fontdimen 2: designsize; % font normal space
fontmaking:=1;
for i:=0 upto 255:
 beginfig(i-256);
  charwd:=charht:=chardp:=charic:=0;
 endfig;
endfor
end.
```

The result of interpreting this program by METAPOST is `cmdfont.tfm`, i.e., a metric file which should be put somewhere where all TFM files reside. It contains 256 characters with all dimensions equal to zero. Most other font parameters are also set to zero. It is obvious that the font design size cannot be zero, but it is not obvious why the width of a space (`fontdimen2`) should be set to the design size. In fact, the actual size is not essential, any non-zero will do. Also, the actual design size value is not important as long as everybody who uses `cmdfont` takes *the same* `designsize` value.

Please note that the `beginfig` parameter is negative. Negative arguments instruct the METAPOST interpreter to output all the EPS files under the same name: `cmdfont.ps`. If we used a seemingly natural form, `beginfig(i)`, our directory would be infested by `cmdfont.0`, `cmdfont.1`, ..., `cmdfont.255` files. We don't need those files and it is easier to throw away one file than 256 files.

Special font in a METAPOST program

Let's trace how the instructions referring to `cmdfont` are parsed by METAPOST. Consider the file named, say, `infont.mp`:

```
beginfig(100);
 draw "META FONT" infont "cmdfont";
 draw "META POST" infont "cmdfont" scaled 2;
endfig;
end.
```

The resulting file, `infont.100`, reads:

```
%!PS
%%BoundingBox: 0 0 0 0
%%Creator: MetaPost
%%CreationDate: 2001.04.13:1950
%%Pages: 1
%*Font: cmdfont 10 10 20:80000000460708
%*Font: cmdfont 20 10 20:80000000440598
%%EndProlog
```

```
%%Page: 1 1
0 0 moveto
(META FONT) cmdfont 10 fshow
0 0 moveto
(META POST) cmdfont 20 fshow
showpage
%%EOF
```

Leaving apart the hairy details of METAPOST-generated PostScript code, let's note that the information about the font `cmdfont` is declared twice in the header of the file, namely, in the lines 6 and 7. Recall that METAPOST strings drawn by an `infont` command are always converted to a *single* `(Postscript string)`, even if they are extremely long. The space inside the METAPOST and the PostScript string denotes the character of code 32. Computer Modern fonts use code 32 for the character *suppress* '˙' used for the letters 'Ł' and 'ł'. Most text fonts use code 32 for a normal non-stretching space.

Consider now a METAPOST program `btexetex.mp` that typesets a text using a construction `btex ... etex`:

```
verbatimtex \font\f cmdfont etex
beginfig(100);
 draw btex \f META FONT etex; draw btex \f META POST etex scaled 2;
endfig;
end.
```

It should not be a surprise that the resulting EPS differs from the previous one:

```
%!PS
%%BoundingBox: 0 0 0 0
%%Creator: MetaPost
%%CreationDate: 2001.04.13:1950
%%Pages: 1
%*Font: cmdfont 10 10 41:8c0e1
%*Font: cmdfont 20 10 41:880b3
%%EndProlog
%%Page: 1 1
0 0 moveto
(META) cmdfont 10 fshow
9.9999 0 moveto
(FONT) cmdfont 10 fshow
0 0 moveto
(META) cmdfont 20 fshow
19.99979 0 moveto
(POST) cmdfont 20 fshow
showpage
%%EOF
```

The difference is that both strings have been split into two pieces (rows 13, 15 and 17, 19). Instead of typesetting a space character (`\char32`), TEX replaced each space with positioning instructions. Wizards able to read the magic hexadecimal sequences occurring in rows 6 and 7 will see that the character of code 32 is missing from the character set used to typeset texts in this particular PostScript file.

The problem of the space will recur in this article.

The special font and TEX+dvips

The files generated with METAPOST are usually included into TEX documents, calendered by dvips and eventually end up in the resulting PostScript file. The only font information TEX needs is the respective metric file (tfm). In contrast, dvips requires for a given font both the tfm file and its glyph shapes. It uses the header of the eps it processes to learn about the character set it needs. Unless we somehow monkey the header, dvips will demand that we provide a bitmap (pk) or a PostScript Type 1 (pfa or pfb) font file. But we have no glyphs for `cmdfont`—neither bitmaps, nor outlines.

We might have used the trick with a virtual font having all characters void, but it wouldn't work—dvips is smart enough to *ignore all texts* typeset with empty characters.

We have found no other way but to choose a popular font and identify it with `cmdfont` by adding an equivalence definition into the `psfonts.map` file. We have chosen `Courier`. The relevant line reads:

```
cmdfont Courier
```

That's all. Now the files typeset with `cmdfont` can be printed as if `cmdfont` was a regular font although the final effect might be weird. However, the `cmdfont` should be used in such a way that a PostScript interpreter would never attempt to display its characters.

How to use cmdfont without external processing

There are two PostScript instructions we have to bridle: `cmdfont` itself and `fshow`. METAPOST typesets the texts using the PostScript instructions with the name derived from the relevant tfm files. The `fshow` command is defined in the file `finclude.pro`. This file is automatically included when dvips encounters the eps file generated by METAPOST containing typeset texts. The piece of METAPOST code quoted below redefines the meaning of both instructions. Our `cmdfont` command interprets the string as an instruction sequence (`cvx exec`) and then it neutralizes the ensuing `fshow`.

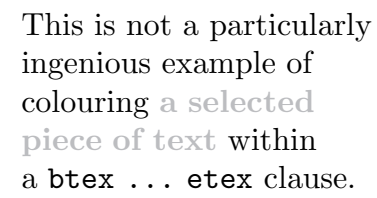
```
def prep_cmdfont =
 special "/fshow where ";
 special " {pop} {/fshow {pop} def} ifelse";
 special "/cmdfont {cvx exec";
```

```
 special "  /fshow.tmp /fshow load def";
 special "  /fshow";
 special "    {pop /fshow /fshow.tmp load def}";
 special "   def";
 special "} def";
enddef;
extra_endfig:=extra_endfig & ";prep_cmdfont;";
```

Using the METAPOST `special` instruction guarantees that the code is moved to the begin of the PostScript code and that's what we wanted to achieve. Note that splitting strings does not bother the '`cvx exec`' doublet.

Employing this technique yields undoubtedly useful results that are rather hard to achieve using "classic" methods. Below, we present three examples of feasible `cmdfont` applications. In the examples we refer to the file named `prepcmdf.mp` containing the definition of `prep_cmdfont` and `extra_endfig` assignment, as described above.

Example 1: Colouring fragments of a TEX text
Let's assume that the METAPOST illustration contains a text with a fragment to be coloured.



The METAPOST source of this illustration is not particularly complicated:

```
input prepcmdf.mp;
verbatimtex
  \def\incmyk#1#2{%
    \leavevmode\rlap{\font\f=cmdfont \f
      gsave #1 setcmykcolor}%
      #2%
      {\font\f=cmdfont \f grestore}}
etex
beginfig(100);
 draw btex \vbox{
   \hsize 40mm \pretolerance10000 \raggedright \noindent
   This is not a particularly ingenious example of colouring
   \incmyk{0 0 0 0.3}{{\bf a selected piece of text}}
   within a~{\tt b{}tex} {\tt ...} {\tt e{}tex} clause.
 } etex scaled 1.2;
endfig;
```

The chief painter is the two-parameter macro `\incmyk` defined in the `verbatimtex ... etex` clause; the `\rlap` instruction used at the beginning of the macro definition

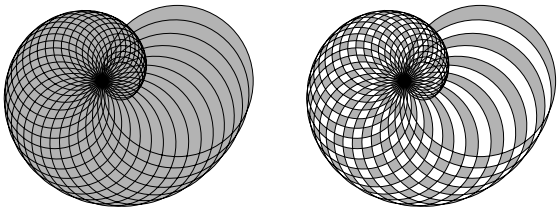
is crucial—we don't want the text "typeset" with `cmdfont` to influence the rest of the typesetting. Recall that the `cmdfont` space has a non-zero width.

This is the piece of the resulting EPS file responsible for the colour changes. The strings fed to `cmdfont` instructions are underlined to improve the legibility.

```
...
(gsave) cmdfont 11.99997 fshow
55.82301 28.69228 moveto
(0) cmdfont 11.99997 fshow
59.8076 28.69228 moveto
(0) cmdfont 11.99997 fshow
63.79219 28.69228 moveto
(0) cmdfont 11.99997 fshow
67.7768 28.69228 moveto
(0.3) cmdfont 11.99997 fshow
71.76138 28.69228 moveto
(setcmykcolor) cmdfont 11.99997 fshow
51.83855 28.69228 moveto
(a) plbx10 11.95514 fshow
62.50629 28.69228 moveto
(selected) plbx10 11.95514 fshow
0 14.3462 moveto
(piece) plbx10 11.95514 fshow
34.15456 14.3462 moveto
(of) plbx10 11.95514 fshow
49.21426 14.3462 moveto
(text) plbx10 11.95514 fshow
73.46477 14.3462 moveto
(grestore) cmdfont 11.99997 fshow
...
```

Example 2: The implementation of `eofill`

PostScript is armed with two basic countour-filling operations: `fill` and `eofill` (even-odd fill). The following picture illustrates the difference. "Snails" are constructed from the circles filled with `fill` (left side) and `eofill` (right side).



Unfortunately, METAPOST uses only `fill`. The `eofill` operator can be implemented using METAPOST `special` instructions. In general, however, it is hairy. Another

solution is the external processing of the resulting EPS files but this is even hairier. The use of `cmdfont` opens a rather simple way to implement `eofill`.

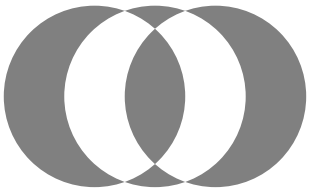
```
1  def eofill(text paths) text modif =
2   begingroup
3    save x_, y_;
4    for p_:=paths:
5     x_:=xpart(llcorner(p_));
6     y_:=ypart(llcorner(p_));
7     exitif true;
8    endfor
9    draw ("/fill.tmp /fill load def " &
10    "/newpath.tmp /newpath load def" &
11    "/fill {/fill{}def /newpath{}def}def")
12    infont "cmdfont" shifted (x_,y_) modif;
13   for p_:=paths: fill p_ modif; endfor
14   draw ("eofill /fill /fill.tmp load def " &
15    "/newpath /newpath.tmp load def")
16    infont "cmdfont"  shifted (x_,y_) modif;
17  endgroup
18 enddef;
```

Just a few words of comment: Lines 9–12 neutralize `fill` and `newpath` instructions appearing in the PostScript code generated by line 13. The code in lines 14–16 invokes `eofill` and restores the meaning of `fill` and `newpath`. `cmdfont` strings need to be positioned in a place that will not change the dimensions of the picture. In this example, strings are put in the lower left corner of the first path from the argument list (lines 4–8). This point satisfies our assumption: all strings have a total width of 0 because spaces are interpreted as characters of code 32.

The macro `eofill` can be used as follows:

```
eofill(
 fullcircle scaled 24mm shifted (-8mm,0),
 fullcircle scaled 24mm,
 fullcircle scaled 24mm shifted (8mm,0))
 withcolor 1/2white;
```

From the user's point of view, there is a difference between the `eofill` operation implemented here and the innate METAPOST `fill` operation—the argument of `eofill` is a list of paths rather than a single path. It is not difficult to predict the result of the code quoted above:



We do not present the PostScript code of the latter example just because boring the reader to death is not exactly our goal. Nevertheless, we recommend to run METAPOST and study the code, it is very instructive reading.

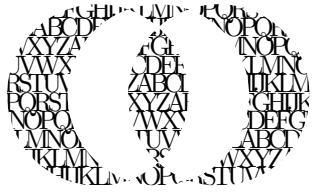
Example 3: The implementation of `eoclip`
The pair of `clip`-`eoclip` operators used for clipping the pictures is analogous to the `fill`-`eofill` pair we considered in the previous example. In particular, METAPOST provides only `clip`. The implementation of `\eoclip` using the `cmdfont` technique is a bit more difficult than that of `eofill`. Here's is our proposal:

```
1  def eoclip(expr pic)(text paths) text modif =
2   begingroup
3    save s_, xmin_, xmax_, ymin, ymax_;
4    xmin_=ymin_:=infinity; xmax_=ymax_:=-infinity;
5    draw ("/clip.tmp /clip load def " &
6     "/newpath.tmp /newpath load def" &
7     "/clip {/clip{}def /newpath{}def}def")
8     infont "cmdfont";
9    picture s_;
10   s_:=image(
11    draw ("eoclip /clip /clip.tmp load def " &
12    "/newpath /newpath.tmp load def")
13    infont "cmdfont"; draw pic);
14   for p_:=paths: clip s_ to p_ modif;
15    if xpart(llcorner(p_ modif)) < xmin_:
16     xmin_:=xpart(llcorner(p_ modif)); fi
17    if xpart(urcorner(p_ modif)) > xmax_:
18     xmax_:=xpart(urcorner(p_ modif)); fi
19    if ypart(llcorner(p_ modif)) < ymin_:
20     ymin_:=ypart(llcorner(p_ modif)); fi
21    if ypart(urcorner(p_ modif)) > ymax_:
22     ymax_:=ypart(urcorner(p_ modif)); fi
23   endfor
24   setbounds s_ to
25     (xmin_,ymin_)--(xmax_,ymin_)--
26     (xmax_,ymax_)--(xmin_,ymax_)--cycle;
27   addto currentpicture also s_;
28  endgroup
29 enddef;
```

The solution we propose is not obvious and has its drawbacks—we would gladly welcome any suggestions how to improve the code. Leaving apart the details, let's concentrate on a few things: (1) Part of the text typeset with `cmdfont` is added to the current picture (`currentpicture` variable, lines 5–8), another part is added to the local picture `s_` (lines 10–13). This is to ensure a proper order of PostScript operations. (2) The text is positioned at the coordinate origin, because eventually the

picture acquires its bounds explicitly (lines 24–26). (3) It resembles more the `eofill` operation defined previously than the original `clip`. We'll need to get used to it...

The illustration below presents the effect of `eoclip`.



It was generated by the following program, again admittedly trivial:

```
beginfig(100);
 picture p; p:=btex \vbox{
  \hsize 45mm \spaceskip-2ptplus1pt \parfillskip0pt
  \baselineskip7pt \lineskiplimit-\maxdimen \noindent
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 } etex;
 eoclip(p)(
  fullcircle scaled 24mm shifted (-8mm,0),
  fullcircle scaled 24mm,
  fullcircle scaled 24mm shifted (8mm,0))
  shifted center p;
endfig;
```

The presented examples are supposed to convince the Reader that the use of `eofill` and `eoclip` is simple. Obviously, it does not imply that the respective definitions are simple. Nevertheless, we think that inventing such definitions is within every METAPOST user's reach. We count on it that METAPOST lovers will develop a heap of useful operations using the described techniques.

The special font and TEX packages

Processing the TEX fragments of a METAPOST source is one of the applications where the `cmdfont` technique proves useful. The basic METAPOST construction, i.e., `btex`

the TEX code `etex` works correctly if we stick to *classic* TEX only. Thus the phrase:

```
verbatimtex \input epsf etex;
picture P; P:=btex \epsfbox{file.eps} etex;
```

does not produce the desired results. The reason is the presence of TEX `\special` commands which (as we mentioned in the introduction) are ignored during the DVI-to-METAPOST transform. The macro `\epsfbox`, defined in the file `epsf.tex` from the DVIPS distribution, analyses the POSTSCRIPT file and generates the appropriate *whatsit* node of type *special*. For instance, the TEX file

```
\input epsf \epsfbox{tiger.ps} \end
```

where `tiger.ps` is a popular *on-duty* EPS file from the GHOSTSCRIPT distribution, produces `\special{PSfile=tiger.ps llx=22 lly=171 urx=567 ury=738 rwi=5450}`. If the TEX fragment is included in a METAPOST file, we should put `\input epsf` inside a `verbatimtex ... etex` block.

As we can see, we cannot move any further without knowing what to do with `\special` instructions. There should be a way to pass the information about them to METAPOST. One such way is redefining the `\special` instruction to transform its content to a string to be typeset with `cmdfont`. The naïve solution

```
\def \special #1{\smash{\hbox{\cmdfont TeXspecial #1}}}
```

does not work properly because the argument of `\special` usually contains spaces which have a non-zero width in `cmdfont`. TEX replaces such spaces by appropriate glues, thus splitting the argument into several substrings that are put into the final POSTSCRIPT file and interwoven with positioning instructions. Moreover, a `\special` argument may contain characters with unexpected categories (such as `$`). Let's assume that `\special` receives its argument as a string of characters of various categories but free from non-expandable non-character tokens (like `\advance`). In theory, such tokens may occur inside `\special`, but we haven't observed any single instance of such an instruction. (It does not mean, however, that they do not exist.) Our task is to convert a string into another string with all categories being "printable".

We propose the following trick: embed the argument of `\special` inside `\csname ... \endcsname`. This way, we get an "error trap" for free because `\csname` crashes when it finds a token that is not a character or a space. The resulting control sequence (with the meaning = `\relax`) can be converted by a `\string` command into a sequence of characters. As every TEX user knows, `\string` expands its arguments into a sequence of characters of category 12... well, not really—spaces get their "traditional" catcode: 10. Therefore all spaces must be converted into explicit `\char32␣` characters by forcing TEX to write the character of code 32 into the DVI file. DVIPS or DVITOMP would convert such a character to an ordinary space. The simplified yet usable file `cmdfont.tex` redefining the `\special` command is given below:

```
\font \cmdfont=cmdfont
% The box keeps the string of characters used instead of \special
\newbox \mpspecialbox
```

```
% We'll keep all specials occurring in the main vertical list
% into the special box.
\newbox \MVLspecialbox
\setbox \MVLspecialbox=\null
\def \mpspecial #1{%
   \setbox \mpspecialbox=\hbox{%
      \cmdfont
      % set escapechar, just in case
      \escapechar=`\\%
% Prepare a macro with the name being the content of our special
% including a backslash at the very beginning.
      \edef \a {\expandafter \string
         \csname TeXspecial: #1\endcsname
         \space \relax}%
      % \b eats the leading backslash, that resulted from
      % \a after it was expanded.
      \def \b ##1{\c}
      % Change every space into the character coded 32.
      \def \c ##1 ##2\relax{##1%
         \ifx $##2$%
         \else \char32 \c##2\relax
         \fi}%
      \expandafter \b \a
   }%

   % If we're in the main vertical list, put the special into
   % a special box, otherwise just typeset it.
   \if \ifvmode\ifinner+\else-\fi\else+\fi +%
       \box \mpspecialbox
   \else
       \global \setbox \MVLspecialbox
            \hbox{\box \MVLspecialbox
                  \kern1sp
                  \box \mpspecialbox}%
   \fi
   }

\def\special{\mpspecial}
```

Such a redefinition of `\special` guarantees that its argument will not be ignored (by DVITOMP) and that METAPOST will receive their string equivalents. More importantly, every such `\special` generates a single string, therefore DVIPS will also generate a single string even if it is enormously large.

What should we do with such strings passed to METAPOST? The final PostScript must be postprocessed with a SED, AWK, or Perl script. This processing is easier than

it could be, because, as we have mentioned, every special is transformed into a single `(Postscript string)`.

PostScript file postprocessing

A small example: `tiger.mp`

```
verbatimtex
  \input cmdfont
  \input epsf
etex

beginfig(100)
  draw btex \epsfxsize=20pt
        \epsfbox{tiger.ps} etex
endfig;
end.
```

The resulting file `tiger.100` contains:

```
%!PS
%%BoundingBox: 0 0 20 21
%%Creator: MetaPost
%%CreationDate: 2001.04.04:1968
%%Pages: 1
%*Font: cmdfont 10 10 20:800277e4000098805748bdc
%%EndProlog
%%Page: 1 1
9.9626 0 moveto
(TeXspecial: PSfile=tiger.ps llx=22 lly=171 u\
rx=567 ury=738 rwi=199) cmdfont
 10 fshow
showpage
%%EOF
```

The parameters `llx`, `lly`, `urx`, and `ury` define the bounding box of the figure to be included; `rwi` is equal to `\epsfxsize` $\times$ 10/`1bp`. We have to replace the string with the code that would be generated by DVIPS if indeed DVIPS found the relevant "real" `\special`.

We have to watch out for strings that begin with `(TeXspecial` and process them up to the final `) cmdfont ... fshow`.

What DVIPS understands

Although the popular DVI-to-PostScript translator called DVIPS understands lots of `\special` patterns, it does not accept all imaginable ones. It can tell friend from foe

by a `\special`'s prefix. Here is the list of the most frequently used prefixes that DVIPS can understand.

- ◇ `papersize`—Defines the page size; METAPOST deals with encapsulated POSTSCRIPT files, thus we can ignore this parameter in most documents as setting page parameters is not allowed in EPS files (according to the Adobe specification of POSTSCRIPT).
- ◇ `landscape`—Specifies page orientation; can be ignored, too.
- ◇ `header`—Adds the specified file to the header of the POSTSCRIPT file made by DVIPS. Keeping in mind that METAPOST files are usually included in TEX documents and processed by DVIPS, we can *clone* this instruction by adding it to an auxiliary TEX file as a *normal* `\special`. If DVIPS felt moved by the standard POSTSCRIPT structured comment `%%DocumentNeededResources`, we could replace the header `\special` by the METAPOST `special`. Actually, DVIPS would just ignore such a comment stolidly.
- ◇ `psfile`—Adds an EPS file. Unfortunately, there's no other way except adding such a file by hand or rather by script. Some non-standard elements like `%*Font` declarations could be cloned to the auxiliary file TEX file.
- ◇ `!`—Its argument is a piece of a literal POSTSCRIPT code. Normally, DVIPS places it in the header of the final output file. Thus, we should clone it to the auxiliary TEX file.
- ◇ `"`—A piece of POSTSCRIPT code, embedded in the graphic environment (coordinate transformation matrix) of an EPS file.
- ◇ `ps:` (note the single colon)—A piece of POSTSCRIPT code embedded in the graphic environment (coordinate transformation matrix) of the POSTSCRIPT file generated by DVIPS. Before and after the code DVIPS adds the positioning instructions.
- ◇ `ps::`, `ps::[begin]`, `ps::[end]`—These constructions are used to concatenate a sequence of `\special` instructions; They are omens of serious trouble. Their description in the DVIPS manual is rather laconic. One can really ƨ¶oil things. We'll assume there are no such specials in the processed files.
- ◇ `em:`—This form was introduced by Eberhard Mattes and used to be understood only by `em`TEX. Although they were quite useful at the time, now they are mostly replaced by pure POSTSCRIPT code. We won't deal with them.
- ◇ `html:`— ... well, perhaps some other time `:-)`.

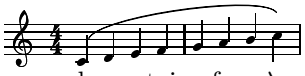
CONSTRUCTION CLONING

Assuming that every METAPOST file eventually gets into TEX and DVIPS, we can save our labour and many opportunities of making errors if we clone some constructions. A `\special` instruction that begins with a `header` or an exclamation mark prefix can be written, as was mentioned, to an auxiliary TEX file to be `\input` again in the final stage of processing. The structured `%*Font` comments can be saved to a pseudo-EPS

file containing only these comments; moreover, an appropriate `\special` command (`\special{psfile ...}`) can be added to the auxiliary TEX file. In this way, the relevant fonts will be included by DVIPS.

The best method to gain some insight into the techniques described herein is to experiment. One can process METAPOST files using the program `despecials` available from `ftp://bop.eps.gda.pl/pub/cmdfont`. It is a PERL script that updates the METAPOST output. It changes `\special` command equivalents expressed by `cmdfont` strings into proper (or sometimes improper) POSTSCRIPT code. Additionally, it produces the TEX file `mpspec.inc` containing selected cloned `\special` commands.

HEADER FILES

Life becomes worse if a TEX file that we add using `btex ... etex` generates `\special` instructions during input. Many macro libraries behave in such a way. As an illustration, let's take a very simple example of the output produced by the LILYPOND program for typesetting music. One of the possible effects of LILYPOND processing its score file is a TEX file (`gamac.tex` in this case) that can be input into the METAPOST figure. Even such a simple example contains four `\special` instructions after conversion to TEX format: two of them are placed in the header part and two are required for slurs. Typesetting scores is, no doubt, one of the most intricate tasks TEX can carry out and there is no way to do it without being *special-infested*. A one-page minuet from the LILYPOND manual contains more than 60 `\special` commands.

Our METAPOST file would look like:

```
verbatimtex
 \input cmdfont
 \input lily-ps-defs
etex
beginfig(100)
 picture P;
 P=btex \input gamac.tex etex;
 draw P;
endfig;
end.
```

Unfortunately, macros contained in the file `lily-ps-defs.tex` generate `\special` commands themselves. Let's consider the result of processing such a file by METAPOST: it generates a file `mpx$$.tex` (the exact name varies from system to system) with an obvious content:

```
\input cmdfont
\input lily-ps-defs
%
\shipout\hbox{\smash{\hbox{\hbox{%
```

```
\input gamac.tex}\vrule width1sp}}}
\end{document}
```

We can see two lines issued by `verbatimtex ... etex`, and the content of the `btex ... etex` block embedded in a rather complicated Russian doll of boxes, to be sent to the DVI file by an explicit `\shipout` command.

Without additional treatment, the effect of TEX processing is a DVI file containing *two pages*. The first one, generated by `\shipout`, will comprise the content of the box *without* the necessary header information. The next page will be generated by the `\end` instruction and it will contain the main vertical list (MVL) with the relevant header information.

There is no unique answer to this problem. It depends on our plans with respect to the header list. One of possible solutions is collecting all `\special` instructions from the main vertical list and adding them to all boxes (or only to the first box) by appropriately redefining the `\shipout` instructions.

Let's assume that the `\special` instructions defined in the `verbatimtex ... etex` block are put into the main vertical list without embedding them in boxes. Under this assumption, it is easy to distinguish a *header*-`\special` from the a *box*-`\special`. The first one is generated in external vertical mode, the latter one in internal vertical mode or horizontal mode (restricted or paragraph). To tell the difference it suffices to use a pair of conditionals `\ifinner` and `\ifvmode`. The macro `\mpspecial` defined in `cmdfont.tex` catches every `\special` that plans to visit the MVL and puts it into a special box, `\MVLspecialbox`. Note that specials are separated by a thin space (1sp), otherwise they would be glued together in the case of an `\unhbox` operation.

If our macros generated `\special` whatsits that are caught by `\mpspecial`, the `\pagetotal` register would be equal to 0pt. If this is not the case, it usually means that something went to the MVL, which is probably wrong. This case cannot be dealt with in a general way but if the total length of the MVL is still less than the page height we can try to pick it up and save into a box:

```
\par
\newbox \MVLbox
\begingroup
   \ifdim \pagetotal>0pt
      \errhelp{I'll save the MVL into MVLbox}
      \errmessage{MVL is not empty}
      \output={\global\setbox
         \MVLbox=\box255}
      \vsize=\pagetotal
      \eject
  \fi
\endgroup
```

It is up to the programmer what the `\MVLbox` is used for, once it is saved.

Recapitulation

Augmenting TEX, METAPOST, METAFONT, and some other programs with a specially treated font looks promising, especially with METAPOST.

Full evaluation of the new possibilities offered by the special font technique requires more experience than we as yet have. The technical details of the `cmdfont` structure need to be worked out. It is not obvious which parameters such a font should have. For instance, which size the space should have: 0pt, 1sp, $\frac{1}{3}$em (a typical value for a text font), or even 1em. We chose the latter because then it is easier to learn, from inside the TEX program, which is the value of the `designsize` parameter. It is also not obvious whether there should be only one `cmdfont` or a whole family of such fonts; and if so, which rules should be applied to avoid a mess.

A mess seems to be a critical threat to the effective application of the techniques described herein. Early standardization, including the font name, the details of the font design and the structure of texts typeset with it, is a *sine qua non* condition of success. We count on the TEX community—without their help it is unlikely that we manage to keep the mess away from this emerging technology which is still in its infancy.

♢ ♢ ♢

MetaType1: *a* METAPOST*-based engine for generating* Type 1 *fonts*

Bogusław Jackowski*, Janusz M. Nowacki†, and Piotr Strzelczyk‡

ABSTRACT.
A package for preparing parameterized outline fonts in PostScript [6] Type 1 [8] format is described. The package makes use of METAPOST [3], awk [4], and T1utils [5], therefore is supposed to be easily portable to various computer platforms. Its beta version along with a sample font (Knuth's logo font) is available from: `ftp://bop.eps.gda.pl/pub/metatype1`

KEYWORDS: outline fonts, scalable fonts, parameterized fonts, PostScript Type 1 fonts, MetaFont, MetaPost

The situation of font collections available for the TEX [1] system can certainly be classified as *bad* if not *ugly*. METAFONT [2], with its bitmap fonts, nowadays seems more and more obsolete. The near future appears to belong to scalable outline fonts. But it would be a pity if METAFONT, with its marvellous engine for creating character shapes, were to remain unused.

Already relatively long ago it was recognized that the design of METAFONT is unsufficiently extensible. In 1989, at the tug meeting, Hobby announced the beginning of work on METAPOST, a program for the generation of a set of eps (*encapsulated PostScript*) files instead of a bitmap font; in 1990, the first version of METAPOST was running. In the same year, Yanai and Berry [23] considered modifying METAFONT in order to output PostScript Type 3 [7] fonts.

Type 3 fonts can be legitimately used with TEX. Actually, bitmap fonts are always implemented as Type 3 fonts by dvi-to-PostScript drivers. Recently, Bzyl [15] has put a lot of effort into the revival of Type 3 fonts in the TEX world. Nevertheless, Type 3 fonts have never become as popular as Type 1 fonts, and probably they never will. One cannot install Type 3 fonts under Windows, MacOS, or X Window, although there are no serious reasons for that—it would suffice to include a PostScript interpreter into an operating system, which is not an unthinkable enterprise. But the commercial world is ruled by its own iffy rights... Anyway, in order to

*B.Jackowski@GUST.org.pl
†J.Nowacki@GUST.org.pl
‡P.Strzelczyk@GUST.org.pl

preserve the compatibility with the surrounding world, one should rather think about Type 1 than Type 3 fonts.

Alas! The issue of converting automatically METAFONT sources to Type 1 format turned out to be more difficult than one could expect (cf. [18, 19, 20, 21, 22]) and after nearly twenty years since the birth of TEX no programming tool for generating Type 1 fonts has appeared. As a consequence there is a glaring scarcity of fonts created by the TEX community.

The MetaType1 package was developed as a response to that bitter situation. Whether it can be classified as *good*—the future will reveal. So far, MetaType1 helped us to prepare a replica of a Polish font designed in the second decade of the twentieth century, Antykwa Półtawskiego [16]. It also proved useful in improving some freely available families of fonts [17].

Which font format?

Among a plethora of currently attainable font formats (see [7] for formats devised by Adobe alone), two are predominant: Type 1 [8] and TrueType [9]. The Type 1 format was Adobe's top secret for six years. In 1990, Adobe decided to disclose the specification after Microsoft and Apple had published the TrueType format. TrueType fonts, despite their very obscure documentation, have become the "mother fonts" of interactive (window) systems. Type 1 fonts can also be used with these systems; however, an additional commercial program, atm (*Adobe Type Manager*), is needed.

From the point of view of TEX users, Type 1 fonts are certainly more suitable, because they are an intrinsic part of the PostScript language. Although a one-to-one conversion between TrueType and Type 1 formats is, in general, impossible, there exist converters that can be used (with care) for this purpose. There are free TrueType-to-Type 1 converters (e.g., [11]), but Type 1-to-TrueType converters seem to be available only as commercial products. Somewhere in between can be located a built-in Windows NT 3.5 converter from Type 1 to TrueType.

Incidentally, contemporary PostScript interpreters accept Type 42 fonts [7], which are essentially TrueType fonts "wrapped" in a PostScript structure. The conversion (one-to-one) between TrueType and Type 42 is pretty simple and free converters are easily available (e.g., [12]).

A few years ago, Microsoft and Adobe announced a joint initiative: they proclaimed that a new font format, OpenType, is to replace both TrueType and Type 1. Microsoft in their documentation on TrueType say, perhaps a bit prematurely, that the TrueType font file specification is "of historical interest only." At present, Adobe offers a cost-free (although licensed) converter from Type 1 to OpenType for the Macintosh and Windows NT platforms. We can expect that more such converters will emerge.

The future is always hidden; we believe, however, that today we can safely invest our efforts in the creation of Type 1 fonts.

Interactive or programming tool?

There are several interactive programs for creating outline fonts. We doubt whether a satisfactorily uniform font can be produced using an interactive tool alone. Fonts are complex monsters and one cannot expect that creating them will ever be an easy task. They are governed by a multitude of parameters such as a stem thickness, serif size and shape, italic angle, the height of majuscules and minuscules, the position of ascenders and descenders, the width of a particular group of characters (e.g., digits should have identical width if we want to use the font in numerical tables), etc. It is particularly difficult to preserve the similarity of shapes appearing repeatedly in a group of characters, e.g., ovals in the letters 'b', 'd', 'o', 'p', and 'q'.

In general, the more irregular the font, the more adequate is an interactive tool. Fonts used for book typesetting, however, are exceptionally uniform. Therefore, some interactive programs provide a programming interface that facilitates controlling uniformness; for example, the commercial FontLab program offers a Python interface in addition. Kinch's approach [18] can be considered as a step further. His MetaFog package is meant for the (semi)manual tuning of glyphs programmed in METAPOST. Despite many advantages, such a "hybrid" approach has a principal drawback: a slight modification of a font may lead to a lot of laborious manual intervention. In particular, parameterization, the boon of the programming approach, is lost.

Only an entirely programmable tool overcomes all these hindrances, but it brings with it its own disadvantages, as programming is apparently difficult for most present-day computer users. This means that the number of MetaType1 users will be limited. Since we are very fond of programming, we can make the prognosis that the number of users will not be less than three.

METAFONT, METAPOST, or ...?

From the very beginning, we abandoned the idea of writing one more stand-alone program (by, e.g., modifying METAFONT or METAPOST) as we didn't want to be involved in technical implementation details. We wanted to make use of existent reliable programs and to focus our attention on the problem of generating Type 1 fonts. Therefore, we had to choose: METAFONT or METAPOST?

The problem with METAFONT is that it writes only gf, tfm, and log files, hence transforming the output from METAFONT to a completely different format, such as PostScript Type 1, is somewhat inconvenient. Its successor, METAPOST, is capable of writing several (text) files, although pictures generated by METAPOST do not form any structure. Fortunately, METAPOST inherited from METAFONT the ability of writing tfm files, which significantly eases the process of generating fonts for TEX, since no extra font installation programs are needed. An argument that can be raised against using METAPOST is that it does not belong to Knuth's canonical distribution; but one cannot avoid using non-canonical software anyway if one wants to produce Type 1 fonts. All in all, we decided to use METAPOST as the bedrock of our package.

Transforming METAPOST output appropriately and imposing a font structure on it has to be done by means of an external program. Our presumption was that it should be a freely available, popular, and portable program.

We believe that AWK (actually, GAWK [13]) meets these requirements. The main drawback of AWK is that it handles only text files. METATYPE1 output files are mostly text files, e.g., AFM (*Adobe font metric*) files, with one pivotal exception, however: many TYPE 1 applications, notably ATM, require a binary form of TYPE 1, PFB (*POSTSCRIPT font binary*). We have considered writing our own assembler in the POSTSCRIPT language (to be used with GHOSTSCRIPT [14]), but finally we gave up and employed a TYPE 1 assembler from the T1UTILS package.

For reasons that are hard to explain, another binary file, PFM (*printer font metric*), is required in order to install TYPE 1 on a Windows system. We decided to prepare our own PERL script for generating PFMs out of AFMs, because the task was relatively simple in spite of the somewhat obscure PFM documentation (sigh) and, moreover, it is easier to manage home-grown software. The script can be used independently of the rest of the engine and, actually, it is not part of the METATYPE1 package.

METATYPE1: AN OVERVIEW

Having answered the fundamental question, i.e., why we are reinventing the wheel, we can start to survey the METATYPE1 engine. Figure 1 shows the structure of the METATYPE1 engine. The boxes drawn with bold lines denote final results, the boxes drawn with dashed lines denote temporary results.

Step 1: METAPOST *processing*
The METAPOST font base contains quite a few macros which are useful in preparing a font. They are similar to Knuth's CMBASE macros. For example, the base contains the macros `beginglyph` and `endglyph`, analogous to `beginchar` and `endchar`. Obviously, the differences are more profound than a simple renaming of some macros, e.g., a lion's share of the code is related to the rules with which TYPE 1 should comply.

From the point of view of a font creator the main difference is that pens are not allowed in METATYPE1 programs. A user is responsible for constructing a proper outline of a glyph: paths should not cross each other (in particular, no self-intersections should occur) and should be properly oriented, i.e., outer paths run anti-clockwise, inner paths run clockwise. There are some macros that facilitate finding the contour of a stroke drawn with a pen ("expanding stroke") or finding the common part of two areas ("removing overlaps"), but, in general, such operations cannot be reliably programmed in METAFONT/METAPOST and therefore users are expected to know what they are doing.

METAPOST works in two passes:

- During the first pass, character glyph files (EPS) and a few auxiliary files (e.g., containing the kerning information) are being generated. The files from the first pass are subsequently processed by AWK.

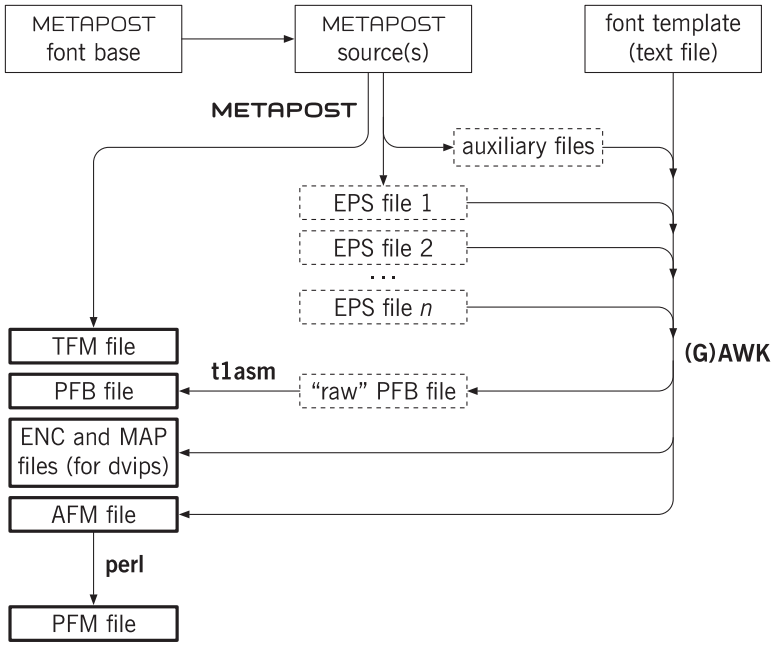


figure 1: The general scheme of the MetaType1 engine.

◇ During the second pass, only TFM files are being generated. All drawing operations are switched off. Writing EPS files, however, cannot be switched off. One can live with it, but it is a somewhat vexing situation—why generate lots of useless files which are only removed later on? The following trick is exploited during the TFM-generating pass: usually, METAPOST appends a numeric extension to the file name; however, if the character code is negative, the extension is simply `ps`; thanks to this, one has to remove only a single dummy file instead of hundreds of them.

Additionally—similar to CMBASE—a user may on demand generate a proof version of character glyphs, e.g., for documentation purposes. Appreciating Knuth's idea of literate programming, we tried to implement it in MetaType1. We wanted METAPOST sources to be simultaneously the ultimate documentation. The MFT utility from the canonical TEX package fits here very well. We slightly enhanced the TEX macros that accompany the program (`mftmac.tex`) in order to facilitate self-documentation. The altered version allows one to include easily a proof version of the glyphs into the formatted source. Figure 2 shows what such a documentation looks like. The displayed sample page is taken from the source of Knuth's LOGO font adapted to MetaType1.

We tried to keep the font base as independent of the Type 1 specification as possible, although, as was mentioned, some peculiarities of Type 1 cannot be ignored. Anyway, we hope that in the (far) future, when Type 1 fonts are finally superseded by a

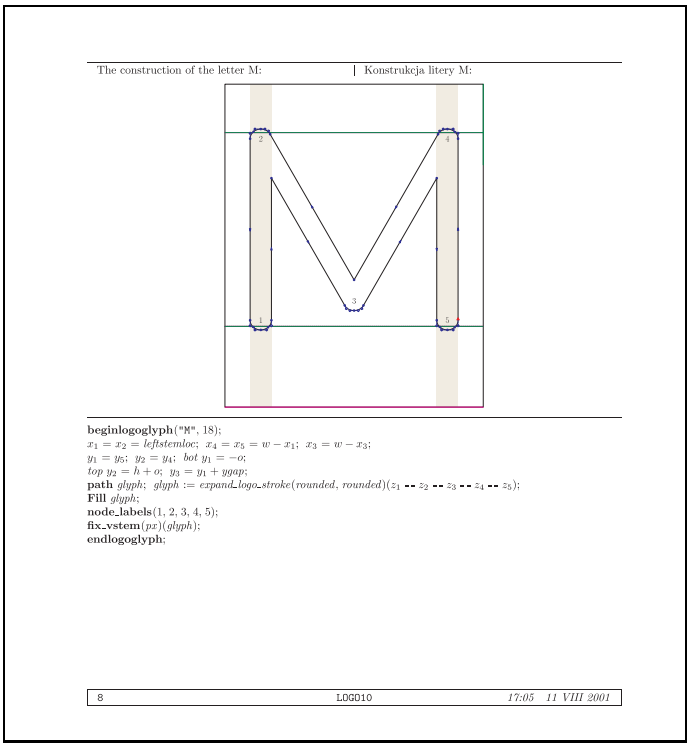


FIGURE 2: AN EXAMPLE OF A SELF-DOCUMENTING METATYPE1 SOURCE.

"Brave New Format," appropriate modification of the base will not be an exceedingly difficult task.

Step 2: AWK *processing*

The main duty of the AWK module is to convert glyphs (EPS) from plain POSTSCRIPT to a form required by the TYPE 1 specification (and accepted by T1UTILS). The TYPE 1 format accepts only integer numbers, therefore rounding is necessary (a user should control the process of rounding at crucial points in METAPOST programs). The only exception is the width of a character. A non-integer width is replaced not by a single number but by a division operation which is allowed in TYPE 1. For example, the width of the character '`T`' from the LOGO10 font is 5.77776 pt, i.e., 577.776 in TYPE 1 grid units; this quantity is represented in the resulting PFB file by '`17911 31 div`' which yields ≈ 577.7742. Appropriate numerators and denominators are computed by means of continued fractions.

In TYPE 1, all dimensions have to be given in relative units, while plain POSTSCRIPT (as output by METAPOST) uses absolute coordinates. Conversion to relative coordinates is also done by AWK.

Perhaps the most complex part of the AWK job is arranging the data properly for hinting. Hints are Type 1 commands that control the discretization of outlines. In a METAPOST source, a user specifies relevant paths and stem values; METAPOST finds acceptable coordinates for the stems and writes this information (embedded as structured comments) into the EPS files. This, however, is not the end of the story. The Adobe Type 1 specification requires that *no hints* of the same kind (horizontal or vertical) *can overlap*. If such a situation occurs, a special routine, called *hint replacement*, should be launched ([8], pp. 69–71). The AWK script does its best to prepare the data properly for the hint replacement process. But it uses some heuristics, therefore the applied algorithm may fail (rather unlikely under normal circumstances).

The result of the AWK run is a disassembled ("raw") PFB file and an AFM file; moreover, ENC and MAP files, to be used with a DVIPS driver, are generated. Optionally, prior to assembling the"raw" PFB file can be processed again by AWK. During this pass, another AWK script is used to search for repeated fragments of code; subroutine definitions are added for such fragments and all their occurrences are replaced by the respective subroutine calls. Usually, this process shortens the PFB file by some 10%. Besides optimization, it also provides an audit of the font, e.g., accented letters should afterwards contain only subroutine calls and no explicit drawing commands.

The AWK stage of font generation is entirely PostScript-oriented. For a different output font format the AWK scripts would have to be rewritten nearly from scratch.

Step 3: Assembling the PFB *file*
This is the simplest step of all: a one-to-one conversion from the disassembled to the final (binary) version of the PFB file is done by a stand-alone freeware utility, T1ASM.

Step 4 (optional): generating the PFM *file*
As was mentioned, PFM files are required if the Type 1 fonts are to be installed on a Windows system. A PFM file contains similar information to that contained in the AFM file, character dimensions, kerning, etc. The main difference is that a PFM file does not contain glyph names, therefore the encoding must be specified in addition. There is a secret byte in a PFM file (85th counting from 0) that contains the relevant information; e.g., the value 238 denotes East European encoding, 206 Central European. Can you guess why? It's simple: $238_{10} = \mathrm{EE}_{16}$, $206_{10} = \mathrm{CE}_{16}$. In order to generate a PFM file conforming to a particular Windows installation, one has to know which number is appropriate. It cannot be excluded that more bombshells of that kind await Windows users. Fortunately, the PERL script is fairly legible and can easily be adjusted if needed.

End or start?

Although we have been working on MetaType1 for a few years, only recently has it stabilized sufficiently to make it available publicly. We must warn potential fearless MetaType1 users, however, that our experience with the package is limited to one complete font (Antykwa Półtawskiego), a few geometric symbol fonts, several improved fonts and one experiment: while preparing this paper we tested MetaType1 against

Knuth's LOGO font. Within *three working days* we were able to modify the METAFONT sources and adjust them to the requirements of METATYPE1. The modified LOGO font is enclosed with the METATYPE1 distribution package. It should be emphasized that the resulting TFM files are 100% compatible with the original ones.

Needless to say, the experiment also unveiled the existence of a few bugs in METATYPE1, both in the METAPOST and AWK parts. Therefore, we consider the public release of METATYPE1 as the start of a new phase rather than the completion of the design process.

Users' feedback cannot be underestimated in this respect. We count upon users' contributions, although we cannot promise a royal road to creating fonts; instead, we can promise satisfaction once a font is ready. We can also assert that programming a font is not reserved for Donald E. Knuth—so, let us go forth now and create masterpieces of digital typography in TYPE 1 format.

REFERENCES

[1] Knuth, D. E., *The TEXbook*. Addison-Wesley, eleventh printing, 1990.

[2] Knuth, D. E., *The METAFONTbook*. Addison-Wesley, seventh printing, 1992.

[3] Hobby, J. D., *The METAPOST Page*. http://cm.bell-labs.com/who/hobby/MetaPost.html

[4] Aho A. V., Kernighan B. W., Weinberger P. J., *The AWK Programming Language*. Addison-Wesley, 1988.

[5] *Type tools*. http://www.lcdf.org/~eddietwo/type/#t1utils

[6] *PostScript Language Reference Manual, 3rd Edition*. http://partners.adobe.com/asn/developer/PDFS/TN/PLRM.pdf

[7] *Adobe Font Formats and File Types*. http://partners.adobe.com/asn/developer/typeforum/ftypes.html

[8] *Adobe Type 1 Font Format*. Addison-Wesley, 1990. http://partners.adobe.com/asn/developer/pdfs/tn/T1_SPEC.PDF

[9] *TrueType Specification, ver. 1.3*. http://www.microsoft.com/typography/tt/ttf_spec/ttspec.zip

[10] *OpenType specification, ver. 1.3 (last update: April 2001)*. http://www.microsoft.com/typography/otspec/otsp13p.zip

[11] *TrueType to PS Type 1 Font Converter*. http://sourceforge.net/projects/ttf2pt1/

[12] Baron, D., *A TrueType to Type 42 Converter*. http://ftp.giga.or.at/pub/nih/ttftot42

[13] *Gnu awk User's Guide*. http://www.gnu.org/manual/gawk/index.html

[14] *Ghostscript, Ghostview and GSview*. http://www.cs.wisc.edu/~ghost/

[15] Bzyl, W., *Reitroducing Type 3 fonts to the TEX world*. Proc. of EuroTEX 2001, 24th–27th September, 2001, Kerkrade, The Netherlands.

[16] Jackowski, B., Nowacki, J. M., Strzelczyk, P., *Antykwa Półtawskiego: A Parameterized Outline Font*. Proc. of EuroTEX'99, 20th–24th September, 1999, Heidelberg, Germany, pp. 109–141.

[17] Jackowski, B., Nowacki, J. M., Strzelczyk, P., *Localizing Type 1 Fonts from Ghostscript Distribution*. BachoTEX'2001, 9th GUST Conference, 29th April–2nd May, 2001, Bachotek, Poland,
`http://www.gust.org.pl/BachoTeX/2001/GSFONTS.PDF`

[18] Kinch, R. J., *MetaFog: Converting METAFONT Shapes to Contours*. TUGboat **16** (3), pp. 233–243, 1995.

[19] Kinch, R. J., *Belleek: A Call for METAFONT revival*. Proc. of 19th Annual TUG Meeting, AuGUST 17–20, 1998, Toruń, Poland, pp. 131–136.

[20] Hoekwater, T., *Generating Type 1 Fonts from METAFONT Sources*. Proc. of 19th Annual TUG Meeting, AuGUST 17–20, 1998, Toruń, Poland, pp. 137–147.

[21] Haralambous, Y., *Parametrization of PostScript Fonts Through METAFONT—an Alternative to Adobe Multiple Master Fonts*. Electronic Publishing, **6** (3), pp. 145–157, 1993.

[22] Malyshev, B. K., *Problems of the Conversion of METAFONT Fonts to PostScript Type 1*. TUGboat, **16** (1), pp. 60–68, 1995.

[23] Yanai, S., Berry D. M., *Environment for Translating METAFONT to PostScript*. TUGboat **11** (4), pp. 525–541, 1990.

♢ ♢ ♢

Natural TEX Notation in Mathematics

Michal Marvan*

abstract. In this paper we introduce Nath, a LaTeX 2.09/2ε style implementing a natural TeX notation for mathematics.

keywords: Natural mathematical notation

The paradigm

The original idea behind TeX was to overcome the principal contradiction of scientific typography, namely that typographers shape publications without understanding their content. But now that the field has been conquered and TeX has become a standard in scientific communication we face an unwanted effect: a decline in typographic quality.

The LaTeX scheme of separating the presentation and content (in the `sty` and `doc` files, respectively) already enabled a basic division of responsibility between scientists and typographers, with a positive impact on the quality of both professional and lay publishing. In contemporary LaTeX we have a rather firmly established content markup for the main parts of a text such as sections, lists, theorems; thanks to styles from the American Mathematical Society we also have a wide variety of mathematical symbols and environments. But the notation for mathematical expressions still encodes presentation.

The basic mathematical constructs of plain TeX refer to presentation by the obvious requirement of universality, and the same holds true for their later reencodings. For instance, `\frac`, which differs from plain TeX's `\over` only by syntax, is a presentation markup to the effect that two elements are positioned one above the other, centered, and separated by a horizontal line. Even though TeX has no technical difficulty typesetting built-up fractions (such as $\frac{A}{B}$) in text style, publishers that still adhere to fine typography may prefer converting them to the slash form A/B. The conversion, during which the mathematical content must not be changed, cannot be reliably done by nonexperts. However, the chance that software can perform the conversion does not turn out to be completely unrealistic, as we shall see below.

Namely, herewith we would like to introduce a *natural mathematical notation* in TeX. By such we mean the coarsest notation for which there exist algorithms that find a typographically sound and mathematically correct context-dependent presentation,

*Mathematical Institute, Silesian University in Opava, Bezručovo nám. 13, 746 01 Opava, Czech Republic. *E-mail*: Michal.Marvan@math.slu.cz

whereas the notation itself is essentially independent of the context. It should be stressed that it is not the goal of natural notation to encode the mathematical content – in contrast to the notation used, e.g., in computer algebra or programming languages.

An accompanying LaTeX 2_ϵ style, Nath, is offered to the public for scrutiny of its strengths and weaknesses, and for practical use. Its mission is to produce traditional mathematical typography with minimal requirements for presentation markup. The price is that TeX spends more time on formatting the mathematical material. However, savings in human work appear to be substantial, the benefits being even more obvious in the context of lay publishing, when expert guidance is an exception.

Preliminaries

Nowadays we recognize two major styles to typeset mathematical material, which we shall call display and in-line. They possibly occur in three sizes: text, script and second level script. The *display* style is characterized by the employment of vertical constructions and arbitrarily sizeable brackets, with emphasis on legibility. Over centuries the display style was commonplace even in in-line formulas, forcing typographers to spread lines as with $\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$. Irregular line spacing and extra costs due to unused white space were among the arguments pushed forward against this practice.

Gradually the *in-line* style evolved, essentially within the Royal Society of London [3, p. 275]. Based on suggestions by the famous logician Augustus de Morgan, the style was introduced by G.G. Stokes [18] and gained strong support from J.W. Rayleigh [15] (all three gentlemen were presidents of the Society). Designed for use under strict limits on the vertical size, the in-line style replaces stacking with horizontal linking, as in $\lim_{n\to\infty}(1+1/n)^n$. Typical is the use of the solidus " / " as a replacement for the horizontal bar of a fraction. The in-line style adds some more ambiguity to that already present in the display style.

Accordingly, Nath uses two distinct math modes, display and in-line, which are fundamentally distinct in TeXnical aspects and go far beyond plain TeX's four math styles (`\displaystyle`, `\textstyle`, `\scriptstyle`, and `\scriptscriptstyle`). The single dollar sign `$` starts the in-line mode; otherwise said, Nath's in-line formulas use in-line mode, and so do sub- and superscripts even within displayed formulas. The double dollar sign `$$` as well as various display math environments start display mode. In contrast to TeX's defaults but in good agreement with old traditions, Nath's default mode for numerators and denominators of displayed fractions is display.

Either mathematical mode can be forced on virtually any subexpression by making it an argument of one of the newly introduced `\displayed` and `\inline` commands. Preserved for backward compatibility, plain TeX's `\displaystyle` and `\textstyle` only affect the size of type, like `\scriptstyle` and `\scriptscriptstyle`. Actually, no such simple switch can alter the fairly sophisticated Nath mode.

Operators

We start with a solution to a subtle problem that occurs in both the display and inline styles, namely, uneven spacing around symbols of binary operations following an operator, as in $\lambda\,\mathrm{id}\,{-}g$. Recall that TEX's capability of producing fine mathematical typography depends on the assignment of one of the eight types (Ord, Op, Bin, Rel, Open, Close, Punct, Inner) to every math atom (see [7, pp. 158 and 170]). Oddly enough, [7, Appendix G, rule 5] says that a Bin atom (a binary operation) preceded by an Op (an operator) becomes an Ord (an ordinary atom like any variable). However, the existence of expressions like $\lambda\,\mathrm{id} - g$ suggests that operators followed by binary operations make perfect sense. Therefore, we propose that the spacing between a Bin atom preceded by an Op be a medium space, i.e., the value in the 2nd row and 3rd column of the table on p. 170 of the TEXbook [7] be '`(2)`' instead of '`*`'. Since TEX provides us with no means to change the table, we had to redefine `\mathop` to a "mixed-type creator," namely `\mathop` from the left and `\mathord` from the right, augmented with appropriate handling of exceptions when Op's behaviour differs from that of Ord. Fortunately, the exceptions occur only when the following atom is Open (an opening delimiter) or Punct (a punctuation), which can be easily recognized by comparison to a fairly short lists of existing left delimiters and punctuation marks. One only must successfully pass over sub- and superscripts as well as over the `\limits` and `\nolimits` modifiers that may follow the Op atom, which on the positive side gives us an opportunity to enable `\\` in Op's subscripts, so that

```
$$
\sum_{i,j \in K \\ i \ne j} a_{ij}
$$
```

prints as

$$\sum_{\substack{i,j\in K\\ i\ne j}} a_{ij}.$$

Another new mixed-type object is `!`, which produces suitable spacing around factorials: `$(m!n!)$` typesets as $(m!\,n!)$. It is simply the exclamation mark (which itself is of type Close) with `\mathopen{}\mathinner{}` appended from the right.

Nath also supports a handy notation for abbreviations in a mathematical formula, such as $\mathrm{e}^{2\pi\mathrm{i}} = -1$, $\mathrm{ad}_x\, y$, $\mathrm{span}\{u, v\}$, $H' = H'_{\mathrm{symm}} + H'_{\mathrm{antisymm}}$, $f|_{\mathrm{int}\,U}$. They are created as letter strings starting from a back quote, e.g., `$‘e^{2\pi‘i}$`, `$‘ad_x y$`, etc.

Fractions

There are three basic types of fractions in modern scientific typography:

$$\text{1) built-up: } \frac{A}{B}, \qquad \text{2) piece: } \tfrac{1}{2}, \qquad \text{3) solidus: } A/B.$$

Mostly they indicate division in a very broad sense; often but not always they can be recast in an alternative form, such as AB^{-1} or $A : B$ (e.g., $\partial f/\partial x$ cannot). Type 1

fractions are now restricted exclusively to display style. The solidus form is mandatory for non-numeric fractions in in-line style; it is also spontaneously preferred in specific situations such as quotient algebraic structures (e.g., $\mathbf{Z}/2\mathbf{Z}$). Type 2 is strictly confined to numeric fractions (when the numerator and denominator are explicit decimal numbers), e.g., $\frac{5}{7}$, $\frac{1}{10\,000}$, $\frac{0.15}{1.22}$. Numeric fractions should be of type 2 or 3 in in-line style. In display style they may occur as both types 1 and 2, depending on the vertical size of the adjacent material. When one changes from display to in-line style, built-up fractions are generally substituted with solidus fractions, and parentheses may have to be added to preserve the mathematical meaning.

Nath supports two commands to typeset fractions: slash `/` and `\frac`. With slash one always typesets a type 3 fraction. With `\frac` one creates a fraction whose type is determined by the following rules: In display style, non-numeric fractions come out as type 1. The type of numeric fractions is determined by the *principle of smallest fences*: A numeric fraction is typeset as a built-up fraction in display style if and only if this will not extend any paired delimiter present in the expression. We explicitly discontinue the tradition according to which numeric fractions adjust their size to the next symbol. For example, Nath typesets

```
$$
(\frac 12 + x)^2 - (\frac 12 + \frac 1x)^2
$$
```

as

$$(\tfrac{1}{2} + x)^2 - \left(\frac{1}{2} + \frac{1}{x}\right)^2.$$

In the sequel, we shall need some definitions. A symbol is said to be *exposed* if it is neither enclosed in paired delimiters nor contained in a sub- or superscript nor used in a construction with an extended line (such as `\sqrt`, `\overline`, or a wide accent). Next, by an element of type Bin* we shall mean an element that either is of type Bin or is of type Ord, starts an expression, and originates from an element of type Bin by [7, Appendix G, rule 5].

In in-line style, the rules that govern typesetting of `\frac` AB are as follows. If A, B are numeric (i.e., strings composed of decimal numbers, spaces and decimal points), then the resulting fraction is of type 2. Otherwise the fraction is of type 3 and bracketing subroutines are invoked. Parentheses are put around the numerator A if A contains an exposed element of type Bin, Rel, Op; or an exposed delimiter that is not a paired delimiter (e.g., /, \ or |). Likewise, parentheses are put around the denominator B if B contains an exposed element of type Bin*, Rel; or an exposed delimiter that is not a paired delimiter. Finally, parentheses are put around the whole fraction if at least one of the columns of Table 1 contains 'Yes' in the corresponding row. For example,

$$\frac{a + \dfrac{b}{b+c}}{1-c} \qquad \to \qquad (a + b/(b+c))/(1-c).$$

Type	Left neighbour	Example	Right neighbour	Example
Ord	Yes[1]	$x(a/b)$	Yes	$(a/b)x$
Op	Yes	$\sin(a/b)$	Yes	$(a/b)\sin x$
Bin*	No[2]	$1+a/b$	No	$a/b+1$
Rel	No	$=a/b$	No	$a/b=$
Open	No	$[a/b$	Yes	$(a/b)[$
Close	Yes	$](a/b)$	No	$a/b]$
Punct	No	$,a/b$	No	$a/b,$
Inner	Yes[1]	$\frac{1}{2}a/b$	Yes	$(a/b)\frac{1}{2}$

[1] No, if the left neighbour is a digit or a piece fraction (hence Inner) and at the same time A starts with neither Bin* nor digit nor decimal point. E.g., $\frac{1}{2}a/b$, but $\frac{1}{2}(-2a/b)$, $\frac{1}{2}(25a/b)$, $\frac{1}{2}(.5a/b)$.

[2] Yes, if A starts with Bin*, e.g., $1+(-a/b)$.

Table 1: Bracketing rules for fractions

Nath's approach to binary operations is mathematically correct under the following assumption: Every binary operator $*$ that occurs in the numerator, denominator, or the immediate vicinity of `\frac` is, similarly to addition, of lower precedence than $/$. An obvious exception is the multiplication "$\cdot$", which is, however, left associative with respect to division and hence $A\cdot B/C = A\cdot(B/C) = (A\cdot B)/C$ anyway. (We also assume that numerators and denominators do not contain exposed punctuation, except for the decimal point.) In particular, Nath converts

$$\frac{A}{B}\otimes\frac{C}{D} \qquad\to\qquad A/B\otimes C/D,$$

and

$$\frac{A\otimes B}{C\otimes D} \qquad\to\qquad (A\otimes B)/(C\otimes D).$$

Literature contains examples of different bracketing, $(A/B)\otimes(B/C)$ and $A\otimes B/C\otimes D$, namely $H_n\big((K/C)\otimes L\big)$ in [11, Ch. V, diag. 10.6] and $\operatorname{Ker}\partial_n/C_n\otimes A$ in [11, Ch. V, proof of Th. 11.1]. Anyway, we feel that giving more binding power to '$\otimes$' than to '$/$' is unfounded.

Now we come to a more delicate question, which reflects a difference between human readability and machine readability. In favour of the former it is often desirable to suppress unneeded parentheses; compare $\exp(x/2\pi)$ and $\exp(x/(2\pi))$. This is one of the reasons why Nath converts

$$\frac{a}{bc} \qquad\to\qquad a/bc$$

and not $a/(bc)$. Here we follow the living tradition according to which a/bc means 'a divided by bc.' Numerous examples of use in professional publications can be easily documented, e.g., [2, p. 9, 34, 52, 89, 115], and the convention is by no means outdated, see, e.g., [14]. It is supported by major manuals of style, albeit by means of examples,

such as $|\langle X_1, X_2\rangle|/\|X_1\|\,\|X_2\|$ in [4]. As much as one chapter in Wick's handbook [20] is devoted to solidus fractions and examples of use; all of them use the same rule as above.

Unfortunately, the convention is not completely devoid of controversy. Some opponents argue that if bc means multiplication, then $a/bc = (a/b)c$ by the current standard rules of precedence, and therefore one should write $a/(bc)$ to have both b and c in the denominator. But, examples like $x/12$, $\partial f/\partial x\,\partial y$, $1/f(x)$, $1/\sin x$, $\mathbf{Z}/2\mathbf{Z}$ show that not every juxtaposition denotes multiplication, while in all of these cases an added pair of parentheses would be certainly superfluous. Understanding juxtaposition requires understanding mathematics, for which reason it is certainly preferable that typography treats all juxtapositions on an essentially equal footing (an exception being subtle rules for close and loose juxtapositions, see the expression $\sin xy \cos xy$ in [1]).

The core of the problem resides in the possible ambiguity of the juxtaposition (see Fateman and Caspi [8], who bring lots of examples of ambiguous notation in the context of machine recognition of TEX-encoded mathematics). However, we feel that by all reasonable criteria, the ambiguity should be kept limited within the denominator, instead of letting it propagate beyond the fraction, which is exactly what would happen if we adapted the competitive rule $a/bc = (a/b)c$. Indeed, the mathematical interpretation of juxtaposition is context dependent, a good case in point being the classic $a(x+y)$. Its meaning depends on whether a is a function that may have $x+y$ as its argument, or not. Under Nath's rules $1/a(x+y)$ is invariably equal to $1/(a(x+y))$, while under the competitive rule the meaning of $1/a(x+y)$ would be $(1/a)(x+y)$ in case of $a = \text{const}$! But then we conclude that the traditional rule $a/bc = a/(bc)$ remains the only reasonable alternative for an unthinking machine.

Anyway, we must admit that there is currently no general consent on this point. The AIP style manual [1] says: "do not write $1/3x$ unless you mean $1/(3x)$," while the Royal Statistical Society [16] considers the notation a/bc "ambiguous if used without a special convention." The *Annals of Mathematical Statistics* even changed its rules from $1/2\pi$ to $1/(2\pi)$ between 1970 and 1971. Use of programming languages and symbolic algebra systems with different syntactic rules also has a confusing effect.

It is certainly true that ambiguity of notation makes reading of mathematical publications more difficult than absolutely necessary. A good solution, which is heartily recommended, amounts to typesetting all difficult fractions in display, or disambiguating them through explicit use of parentheses or otherwise.

Nath's solution is, we believe, the best possible from those available, given the fact that TEX does not provide tools for recognizing close (spaceless) juxtaposition. Nath essentially treats juxtaposition as a binary operation of higher precedence than solidus (even the loose juxtaposition expressed via a small amount of white space, such as `\thinmuskip` around Op's):

$$\frac{1}{\cos x} \qquad \to \qquad 1/\cos x.$$

The only exception is that the right binding power of loose juxtaposition is considered uncomparable to the left binding power of the solidus, so that, e.g., $\sin x/y$ comes out

Left delimiters		Right delimiters	
`(`	(	`)`	)
`[`,`\lbrack`	[	`]`,`\rbrack`	]
`\{`, `\lbrace`	{	`\}`, `\rbrace`	}
`<`, `\langle`	⟨	`>`, `\rangle`	⟩
`\lfloor`	⌊	`\rfloor`	⌋
`\lceil`	⌈	`\rceil`	⌉
`\lvert`, `\left|`	|	`\rvert`, `\right|`	|
`\lBrack`, `\double[`	⟦	`\rBrack`, `\double]`	⟧
`\lAngle`, `\double<`	⟪	`\rAngle`, `\double>`	⟫
`\lFloor`	⦋	`\rFloor`	⦌
`\lCeil`	⦍	`\rCeil`	⦎
`\lVert`, `\ldouble|`	‖	`\rvert`, `\rdouble|`	‖
`\triple[`	⦃	`\triple]`	⦄
`\triple<`	⦉	`\triple>`	⦊
`\ltriple|`	⦀	`\rtriple|`	⦀

Table 2: Paired delimiters

as truly ambiguous (following [1]); hence Nath converts

$$\sin \frac{x}{y} \qquad \to \qquad \sin(x/y)$$

and

$$\frac{\sin x}{y} \qquad \to \qquad (\sin x)/y$$

– even though Wick interprets $\sin x/y$ as $(\sin x)/y$.

Delimiters

Plain TEX introduces various delimiter modifiers such as `\left` and `\right`. If used continually without actual need, as is often done, they produce unsatisfactory results; such continual use is as undesirable as is the failure to use them when they are actually needed. Under natural notation every left parenthesis is a left delimiter by default, and Nath does its best to ensure proper matching to whatever is enclosed.

Table 2 lists paired delimiters. Their presentation depends on the current mode. In display mode delimiters automatically adjust their size and positioning to match the material enclosed (thus rendering `\left` and `\right` nearly obsolete), and do so across line breaks (which themselves are indicated by mere `\\` whenever allowed by the context). Around asymmetric formulas the delimiters may be positioned asymmetrically.

A particularly nice example, taken from [9, p. 4], is

$$\frac{M}{\left(1-\dfrac{x_1+\cdots+x_n+pZ}{r}\right)\left(1-p\dfrac{\dfrac{\partial Z}{\partial x_2}+\cdots+\dfrac{\partial Z}{\partial x_n}}{\rho}\right)}$$

(no modifiers in front of the parentheses).

The modifiers `\double` and `\triple` create double and triple counterparts of delimiters, such as

$$\left[\!\left[\frac{x}{y}\right]\!\right].$$

We also introduce *middle delimiters*: `\mid` and `\middle|` produce | , `\Mid` and `\double|` produce ‖ , and `\triple|` produces ||| . They have exactly the size of the nearest outer pair of delimiters. For example:

$$\Big\{(x_i)\in R^\infty \,\Big|\, \sum_{i=1}^{\infty} x_i^2 = 1\Big\}.$$

Observe that matching is done in a subtle way, disregarding sub- and superscripts, accents, and other negligible parts. (Let us also note that in order to implement the above-mentioned principle of smallest fences in display style, Nath represents numeric fractions as middle delimiters.)

With nested delimiters it is frequently desirable that the outer delimiters are bigger than the inner ones. In displayed formulas this is controlled by a count `\delimgrowth` that when set to n makes every nth delimiter bigger. One should set `\delimgrowth=1` when a display contains many vertical bars:

$$C_6\left|\left|f\int_\Omega\left|\tilde{S}_{a,-}^{-1,0}W_2(\Omega,\Gamma_1)\right|\right|\,|u|\to W_2^{\tilde{A}}(\Omega;\Gamma_r,T)\right|\right|.$$

(cf. [17]).

In in-line mode a completely different mechanism is needed, which would be applicable to implicit delimiters introduced by `\frac`. We introduce a *command* `\big` having the effect that the next entered level of delimiters will be set in big size (in plain TEX's sense). For instance, `$\Delta \big \frac 1{f(x)}$` produces $\Delta\big(1/f(x)\big)$. It is an error to place a `\big` within delimiters that are not big. Observe that Nath's `\big` need not immediately precede a delimiter; this gave us an opportunity to introduce `\bigg` as an abbreviation for `\big\big`.

Unbalanced delimiters may be present in an in-line formula (as is often the case in indices in differential geometry), but then cannot be resized.

Displayed formulas

Displayed formulas have never been a serious problem. Yet there is room for innovation and simplification of the current presentation markup. Downes presented a style [6], which breaks multiline displayed equations automatically. With Nath, every end of line must be marked by an explicit `\\`, but this `\\` can be used at almost any place where it makes sense. In particular, `$$` $\cdots$ `=` $\cdots$ `\\` $\cdots$ `=` $\cdots$ `$$` is a valid syntax. The result is a multiline formula without special alignment:

$$\begin{aligned}\qquad\qquad &= \qquad\qquad\qquad\qquad\qquad\qquad\\ \qquad\qquad\qquad\qquad\qquad\qquad &= \qquad\qquad\end{aligned}$$

(by default, Nath indents all displayed equations by `\mathindent=4pc`). Within the `equation` environment, the formula obtains a single centered number.

A kind of alignment can be obtained with the wall–return construction. The syntax is `\wall` $\cdots$ `\\` $\cdots$ `\\` $\cdots$ `\\` $\cdots$ `\return`, and can be nested. Here is an example

```
$$
\stuff \wall = \stuff + (\wall \stuff \\
                                \stuff)
                          \return
             = \stuff
       \return
$$
```

gives

$$\begin{aligned}\qquad\qquad &= \qquad\qquad + (\qquad\qquad\qquad\qquad\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad)\\ &= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad.\end{aligned}$$

The meaning is that the typeset material must not cross the wall.

Display delimiters cannot be combined with alignments unless every cell has balanced delimiters, which is certainly the case with matrices, but not necessarily with other alignment environments, such as `eqnarray`. The purpose of these environments is, essentially, to align binary relation symbols in one of the two typographically relevant situations:

(1) an n-line sequence of equalities and relations;

(2) a pile of n distinct formulas.

In case (1), walls alone represent a simple alternative that spares the $2n$ alignment symbols `&` required by eqnarray. It is also possible to put a wall–return block into one cell of an alignment.

Acknowledgements

I would like to thank many people, explicitly Eylon Caspi, Michael Guravage, Wietze van der Laan, Petr Sojka, Simon Pepping, Alex Verbovetsky, and staff in the National Library of Scotland in Edinburgh.

REFERENCES

[1] *AIP Style Manual*, 4th edition (Amer. Inst. Physics, New York, 1990).

[2] S.H. Benton, *The Hamilton–Jacobi Equation. A Global Approach* (Academic Press, New York, 1977).

[3] F. Cajori, *A History of Mathematical Notations I, II* (Open Court, Chicago, 1928, 1929).

[4] *The Chicago Manual of Style* (The Univ. Chicago Press, Chicago and London, 1969).

[5] A. De Morgan, The calculus of functions, in: Encyclopaedia Metropolitana (1845).

[6] M. Downes, Breaking equations, *TUGboat* 18 (1997) 182–194.

[7] D.E. Knuth, *The TEXbook* (Addison Wesley, Reading, 1984)

[8] R.J. Fateman and E. Caspi, Parsing TEX into mathematics, poster presented at International Symposium on Symbolic and Algebraic Computation (ISSAC '99), Vancouver BC Canada, July 28–31, 1999.

[9] E. Goursat, *Leşons sur l'Intégration des Équations aux Dérivées Partielles du Premier Ordre*, 2nd ed. (Hermann, Paris, 1921)

[10] J. Larmor, Practical suggestions on mathematical notation and printing, an appendix to [15].

[11] S. Mac Lane, *Homology* (Springer, Berlin, 1967).

[12] A. Macfarlane, Sir George Gabriel Stokes, in: *Lectures on Ten British Physicists of the Nineteenth Century* (New York, 1919).

[13] M. Marvan, Přirozená matematická notace v TEXu, in: J. Kasprzak and P. Sojka, eds., *SLT 2001* (Konvoj, Brno, 2001) 53–59.

[14] Mathematical Composition Setters, Sample style setting, online www.mcs-ltd.co.uk, accessed Jan. 8, 2001.

[15] J.W. Rayleigh, Address of the president, Lord Rayleigh, O.M., D.C.L., at the anniversary meeting on November 30, 1918, *Proc. Roy. Soc. London, Sect. A* 82 (1909) 1–17.

[16] Royal Statistical Society, Notes for the preparation of mathematical papers, *J. Roy. Stat. Soc., Ser. A* 136 (1973) 293–294.

[17] A. Samarin, F. Mittelbach and M. Goossens, The LATEX Companion (Addison-Wesley, Reading, 1994).

[18] G.G. Stokes, *Mathematical and Physical Papers, Vol. I* (Cambridge Univ. Press, Cambridge, 1880).

[19] E. Swanson, *Mathematics into Type* (Amer. Math. Soc., Providence, 1987).

[20] K. Wick, *Rules for Typesetting Mathematics* (Mouton, The Hague, 1965); translated from the Czech original *Pravidla Matematické Sazby* (Academia, Praha, 1964).

□ ◇ □

TEX in Teaching

Michael Moortgat, Richard Moot & Dick Oehrle*

abstract. A well-known slogan in language technology is 'parsing-as-deduction': syntax and meaning analysis of a text takes the form of a mathematical proof. Developers of language technology (and students of computational linguistics) want to visualize these mathematical objects in a variety of formats.

We discuss a language engineering environment for computational grammars. The kernel is a theorem prover, implemented in the logic-programming language Prolog. The kernel produces latex source code for its internal computations. The front-end displays these in a number of user-defined typeset formats. Local interaction with the kernel is via a tcl/tk GUI. Alternatively, one can call the kernel remotely from dynamic PDF documents, using the form features of Sebastian Rahtz' `hyperref` package.

This paper discusses some uses of the dynamic possibilities offered by Sebastian Rahtz' `hyperref` package in the context of a courseware project we have been engaged in. The project provides a grammar development environment for Type-Logical Grammar — one of the formalisms that are currently used in computational linguistics. Our paper is organized as follows. First, we offer the reader a glimpse of what type-logical grammars look like. In the next section, we discuss the TEX-based visualisation tools of the Grail workbench in its original unix habitat. Finally, we report on our current efforts to provide browser-based access to the Grail kernel via dynamic PDF documents.

Type-logical grammar

Type-logical (*TLG*) grammar is a logic-based computational formalism that grew out of the work of the mathematician Jim Lambek in the late Fifties. For readers with easy access to issues of the *American Mathematical Monthly* in the pre-TEX era, the seminal paper (Lambek 1958) is warmly recommended; (Moortgat 1997) gives an up-to-date survey of the field. The mathematically-inclined reader of these Proceedings will easily appreciate why it is such a pleasure to work with TLG.

As the name suggests, *TLG* has strong type-theoretic connections. One could think of it as a functional programming language with some special features to handle the

*We thank Willemijn Vermaat and Bernhard Fisseni for helpful discussions of TEXnicalities. For comments and suggestions, contact {moortgat,oehrle}@let.uu.nl.

peculiarities of natural (as opposed to programming) languages. In a functional language (say, Haskell), expressions are typed. There is some inventory of basic types (integers, booleans, ...); from types T, T' one can form functional types $T \to T'$. With these functional types, one can do two things. An expression/program of type $T \to T'$ can be used to compute an expression of type T' by *applying* it to an argument of the appropriate type T. Or a program of type $T \to T'$ can be obtained by *abstracting* over a variable of type T in an expression of type T'. Below we give a simple example: the construction of a square function out of a built-in times function. We present this as a logical derivation — the beautiful insight of Curry allows us to freely switch perspective between types and logical formulas, and between type computations and logical derivations in a constructive logic (Positive Intuitionistic Logic).

$$\dfrac{\dfrac{\dfrac{\mathsf{times} : \mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int}) \quad x : \mathsf{Int}}{(\mathsf{times}\ x) : \mathsf{Int} \to \mathsf{Int}}\ (\mathit{Elim} \to) \qquad x : \mathsf{Int}}{(\mathsf{times}\ x\ x) : \mathsf{Int}}\ (\mathit{Elim} \to)}{\lambda x.(\mathsf{times}\ x\ x) : \mathsf{Int} \to \mathsf{Int}}\ (\mathit{Intro} \to)$$

How can we transfer these ideas to the field of natural language grammars? The basic types in this setting are for expressions one can think of as 'complete' in some intuitive sense — one could have a type np for names ('Donald Knuth', 'the author of *The Art of Computer Programming*', ...), common nouns n ('author', 'art', ...), sentences s ('Knuth wrote some books', 'TEX is necessary', ...). Now, where a phrase-structure grammar would have to add a plethora of non-terminals to handle incomplete expressions, in *TLG* we use functional (implicational) types for these. A determiner like 'the' is typed as a function from n expressions (like 'author') to np expressions; a verb phrase (like 'is necessary') as a function from np expressions into s expressions, and so on.

To adjust the type-logical approach to the natural language domain, we have to introduce two refinements. The syntax of our programming language example obeys the martial law of Polish prefix notation: functions are put before their arguments. Natural languages are not so disciplined: a determiner (in English) comes before the noun it combines with; a verb phrase follows its subject. Instead of one implication, *TLG* has two to capture these word-order distinctions: an expression of type T/T' is *prefixed* to its T'-type argument; an expression $T'\backslash T$ is *suffixed* to it. An example is given below. (The product $\circ$ is the explicit structure-building operation that goes with use of the slashes. It imposes a tree structure on the derived sentence.)

$$\dfrac{\mathsf{mathematicians} \vdash np \qquad \dfrac{\mathsf{like} \vdash (np\backslash s)/np \quad \mathsf{TEX} \vdash np}{\mathsf{like} \circ \mathsf{TEX} \vdash np\backslash s}\ [/E]}{\mathsf{mathematicians} \circ (\mathsf{like} \circ \mathsf{TEX}) \vdash s}\ [\backslash E]$$

The second refinement has to do with the management of 'programming resources'. In our Haskell-style example, one can use resources as many times as one wants (or not use them at all). You see an illustration in the last step of the derivation, where two occurrences of $x : \mathsf{Int}$ are withdrawn simultaneously. In natural language, such a

cavalier attitude towards occurrences would not be a good idea: a well-formed sentence is not likely to remain well-formed if you remove some words, or repeat some. (You will agree that 'mathematicians like' does not convey the message that mathematicians like mathematicians.) Our grammatical type-logic, in other words, insists that every resource is used exactly once. And in addition to resource-sensitivity, there may be certain structural manipulations that are allowable in one language as opposed to another. To control these, there is a module of non-logical axioms (so-called structural postulates) in addition to the logical rules for the slashes. The derivation below contains such a structural move: the inference labeled $P2$ which uses associativity to rebracket the antecedent tree.

$$\dfrac{\dfrac{\mathsf{the}}{np/n} \quad \dfrac{\dfrac{\mathsf{book}}{n} \quad \dfrac{\dfrac{\mathsf{that}}{(n\backslash n)/(s/np)} \quad \dfrac{\dfrac{\dfrac{\mathsf{knuth}}{np} \quad \dfrac{\dfrac{\mathsf{wrote}}{(np\backslash s)/np} \quad [\mathsf{p}_1 \vdash np]^1}{\mathsf{wrote} \circ \mathsf{p}_1 \vdash np\backslash s}\,[\backslash E]}{\mathsf{knuth} \circ (\mathsf{wrote} \circ \mathsf{p}_1) \vdash s}\,[P2]}{(\mathsf{knuth} \circ \mathsf{wrote}) \circ \mathsf{p}_1 \vdash s}\,[/I]^1}{\mathsf{knuth} \circ \mathsf{wrote} \vdash s/np}\,[/E]}{\mathsf{that} \circ (\mathsf{knuth} \circ \mathsf{wrote}) \vdash n\backslash n}\,[\backslash E]}{\mathsf{book} \circ (\mathsf{that} \circ (\mathsf{knuth} \circ \mathsf{wrote})) \vdash n}\,[/E]}{\mathsf{the} \circ (\mathsf{book} \circ (\mathsf{that} \circ (\mathsf{knuth} \circ \mathsf{wrote}))) \vdash np}\,[/E]$$

At this point, you are perfectly ready to write your first type-logical grammar! Assign types to the words in your lexicon, and decide whether any extra structural reasoning is required. The type-inference machine of *TLG* does the rest.

The Grail theorem prover

The Grail system, developed by the second author, is a general grammar development environment for designing and prototyping type-logical grammars. We refer the reader to (Moot 1998) for a short description of the system, which is available under the GNU General Public License agreement from `ftp.let.uu.nl/pub/users/moot`. The original Grail implementation presupposes a unix environment. It uses the following software components:

- SICStus Prolog: the programming language for the kernel;
- Tcl/Tk for the graphical user interface;
- a standard teTeX environment for the visualization/export of derivations.

In a Grail session, the user can design a grammar fragment, which in the *TLG* setting comes down to the following:

- assign formulas (and meaning programs) to words in the lexicon or edit formulas already in the lexicon,
- add or modify structural rewrite rules,

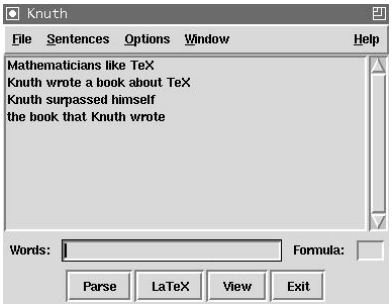


FIGURE 1: THE GRAIL MAIN WINDOW

- and finally, to run the theorem prover on sample expressions to see which expressions are grammatical in the specified grammar fragment by trying to find a derivation for them.

The theorem prover can operate either automatically or interactively. In interactive mode, the user decides which of several possible subproofs to try first, or to abandon subproofs which the user knows cannot succeed, even though the theorem prover might take a very long time to discover that. Another possibility is that the user is only interested in some of the proofs. The interactive debugger is based on proof net technology — a proof-theoretic framework specially designed for resource-sensitive deductive systems. In Figure 1, we give a screenshot of the main window of the GUI. Figure 2 shows a proof net for the derivation of the sentence 'Knuth surpassed himself'.

When successful derivations have been found, GRAIL can convert the internal representation of the proof objects to natural deductions in the form of LATEX output. We have already seen some examples in the previous section. Though the internal representation of derivations contains a lot of information, the structure is basically simple: a proof object consists of a conclusion together with a list of proof objects

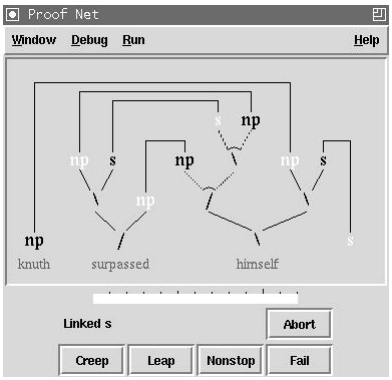


FIGURE 2: THE PROOF NET DEBUGGER WINDOW

$$\dfrac{\dfrac{\mathsf{knuth}}{np} \quad \dfrac{\dfrac{\mathsf{surpassed}}{(np\backslash s)/np} \quad \dfrac{\mathsf{himself}}{((np\backslash s)/np)\backslash(np\backslash s)}\,[\backslash E]}{\mathsf{surpassed} \circ \mathsf{himself} \vdash np\backslash s}\,[\backslash E]}{\mathsf{knuth} \circ (\mathsf{surpassed} \circ \mathsf{himself}) \vdash s}$$

FIGURE 3: PRAWITZ STYLE NATURAL DEDUCTION OUTPUT

1.	$\mathsf{knuth} : np - \mathbf{knuth}$	Lex
2.	$\mathsf{surpassed} : (np\backslash s)/np - \mathbf{surpass}$	Lex
3.	$\mathsf{himself} : ((np\backslash s)/np)\backslash(np\backslash s) - \lambda z_2.\lambda x_3.((z_2\ x_3)\ x_3)$	Lex
4.	$\mathsf{surpassed} \circ \mathsf{himself} : np\backslash s - \lambda x_3.((\mathbf{surpass}\ x_3)\ x_3)$	$\backslash E\ (2,3)$
5.	$\mathsf{knuth} \circ (\mathsf{surpassed} \circ \mathsf{himself}) : s - ((\mathbf{surpass\ knuth})\ \mathbf{knuth})$	$\backslash E\ (1,4)$

1. ((**surpass knuth**) **knuth**)

FIGURE 4: FITCH STYLE NATURAL DEDUCTION OUTPUT

which validate this conclusion and the LATEX output is produced by recursively traversing this structure.

A number of parameters guide the production of the LATEX proofs. The output parameters include, for example, a choice to have proofs presented in the tree-like Prawitz output format, as shown in Figure 3, or in the list-like Fitch output format, as shown in Figure 4. The Fitch list format is handy when the user chooses to include the meaning assembly in a derivation: tree format quickly exceeds the printed page format in these cases. The Prawitz derivations are typeset in LATEX using the `proof.sty` package of Tatsuta (1997), but, as the conversion to LATEX is quite modular, it would be possible to generate proofs in different formats, using for example the `prooftree.sty` of Taylor (1996) as an alternative.

An extract of the LATEX source for Figure 3 is shown in Figure 5. Automated generation of derivations is agreeable especially in the case of more complex examples, which can be frustrating to produce or edit manually.

```
\infer[\bo \bs E \bc^{}]
   {\textsf{knuth}\circ_{}(\textsf{surpassed}\circ_{}\textsf{himself})\vdash s}{
   \infer{np}{\textsf{knuth}}}
&
   \infer[\bo \bs E \bc^{}]
      {\textsf{surpassed}\circ_{}\textsf{himself} \vdash np \bs_{}s}{
      \infer{(np \bs_{}s) /_{}np}{\textsf{surpassed}}}
   &
      \infer{((np \bs_{}s) /_{}np) \bs_{}(np \bs_{}s)}{\textsf{himself}}
   }
}
```

FIGURE 5: THE LATEX SOURCE FOR FIGURE 3

GRAIL ON THE WEB

The Tcl/Tk-based graphical interface to GRAIL provides a pleasant working environment, especially for users unfamiliar with Prolog. But it is a complex platform, dependent on the interaction of a number of programs—Prolog, LATEX, Tcl/Tk. The World Wide Web provides an environment that in principle allows access to GRAIL's facilities for grammatical research, testing, and development to anyone with a graphical browser, and this is the objective of our current efforts. Moving GRAIL onto the web involves a natural series of stages, which we describe below.

Command line interaction

It would be difficult to manage via a browser the interaction of a remote user and the Tcl/Tk graphical interface. But it is not so difficult to manage this interaction directly via the SICStus Prolog command line prompt. In particular, if one wants to test whether a given expression is assignable a particular type relative to a particular fragment, it is enough to start GRAIL under SICStus, load the fragment in question—simply an additional sequence of Prolog clauses—then pass the expression and the goal formula to GRAIL. The results of the parse can be written out as a LATEX file and displayed in .dvi or .ps or .pdf format, as discussed above.

One commonly executes these steps sequentially, as shown below, suppressing unnecessary detail and extraneous messages. The command line instruction `% sicstus` initiates a session with SICStus Prolog and the command `consult('notcl2000.pl')` loads GRAIL without the Tcl/Tk interface. One can then load a fragment—here `consult('knuth.pl')` —and test whether the expression Knuth surpassed himself can be assigned the type s by entering the clause `tex([knuth,surpassed,himself],s).`

• • •

```
% sicstus
SICStus 3.8.5 (sparc-solaris-5.7): Fri Oct 27 10:12:22 MET DST 2000
Licensed to let.uu.nl
| ?- consult('notcl2000.pl').
{consulting notcl2000.pl...}
========================
= Welcome to Grail 2.0 =
========================
{Warning: something appears to be wrong with the TclTk library!}
{You can still use Grail, but you will have limited functionality}

yes

| ?- consult('knuth.pl').
{consulting knuth.pl...}
{consulted knuth.pl in module user, 20 msec 6952 bytes}

yes
```

```
| ?- tex([knuth,surpassed,himself],s).

===
[knuth,surpassed,himself] => s
===
Lookup: 0

Max # links: 12
===

(FAILED). surpass(knuth,knuth)

(knuth *[] (((G \[] (E *[] (surpassed *[] G))) /[] E) *[] himself))-->>
IRREDUCIBLE

===

1. surpass(knuth,knuth)

(knuth *[] (((G \[] (G *[] (surpassed *[] E))) /[] E) *[] himself))-->>
(knuth *[] (surpassed *[] himself))

===

Telling LaTeX output directory eg.tex

1 solution found.
CPU Time used: 0.200

Telling LaTeX output directory eg.tex

true ? latex ready
```

• • •

The final comments indicate that GRAIL has written out the proof to the file `eg.tex` in a way that can be inserted directly in a LATEX document, as we have seen in Figure 5.

Shell interaction

SICStus provides facilities to combine all the steps of program initiation and input just illustrated into a single command line, using the built-in SICStus predicates. During a SICStus session, a call to the predicate `save_program` saves the state of the run of SICStus in a way that allows it to be restarted at exactly the same point.

For instance, if a SICStus program contains the clause

```
make_savedstate:- save_program(wwwgrailstate, startup).
```

the run state will be saved as the executable *wwwgrailstate*, and upon reinitiation will attempt to prove the predicate `startup/0`. To restart SICStus in this way, one calls SICStus with the *-r* flag:

```
% sicstus -r wwwgrailstate
```

Finally, there is an additional flag which allows one to pass a sequence of arguments to the re-initiated state, bound as a list of Prolog atoms to the special built-in constant *argv*, as shown below.

```
% sicstus -r wwwgrailstate -a arg1 arg2 ...
```

Now, for our purposes, these arguments can provide information about a particular fragment to be loaded, a variety of choices about proof display, etc., and finally, the goal formula of the expression to be tested and the list of words making up the expression itself. What remains is to unpack the list *argv* inside and redeploy these individual arguments appropriately.

Here is an example in which the first argument following the *-a* flag selects the fragment *knuth.pl*, the second through the fifth arguments set switches governing the proof format, the sixth argument (*s*) sets the goal formula, and the remaining arguments specify the list of words of the expression to be tested.

```
% sicstus -r wwwgrailstate
-a knuth yes yes yes inactive nd s knuth surpassed himself
{restoring wwwgrailstate...}
{wwwgrailstate restored in 80 msec 513808 bytes}
{consulting knuth.pl...}
{consulted knuth.pl in module user, 20 msec 7064 bytes}

===
[knuth,surpassed,himself] => s
===

Lookup: 0

Max # links: 12
===

(FAILED). surpass(knuth,knuth)

(knuth *[] (((G \[] (E *[] (surpassed *[] G))) /[] E) *[] himself))-->>
IRREDUCIBLE

===

1. surpass(knuth,knuth)
```

```
(knuth *[] (((G \[] (G *[] (surpassed *[] E))) /[] E) *[] himself))-->>
(knuth *[] (surpassed *[] himself))

===

1 solution found.
CPU Time used: 0.200
```

• • •

Although we will not discuss here how the list of arguments bound to *argv* is treated internally to the SICStus code in *wwwgrailstate*, the reader may observe the similarity of the standard error message printed out immediately above and the standard error message encountered earlier in the course of our interactive command-line session with GRAIL.

Web interaction

From this point, it is straightforward to lift the interaction to the web. The arguments are encoded in an active form (or simply passed directly using the standard URL syntax for CGI programming). The form document can be prepared as an HTML document or as an active PDF document, using Sebastian Rahtz's hyperref package and Han The Thanh's pdfLATEX program. We will come back to details of this step momentarily, after looking briefly at how the server is set up to deal with such an interaction.

Lincoln Stein's Perl module CGI.pm makes it especially simple to grab these arguments, check them for correctness ('untainting'), and pass them to the saved SICStus state. And the parse can be written out in LATEX, then passed back to the user's browser as a .pdf file (using pdfLATEX).

Fragment display

A disadvantage of the above setup is that the end user must have a reasonable idea of the capabilities of each fragment: what is its lexicon? what kinds of grammatical questions does it address? As an aid to the user, it would be helpful to display the fragment in a pleasant form, providing all the information the user needs. GRAIL already has facilities to spell out the properties of fragments in LATEX—including postulates, lexicon, and stored examples. One can collect these as a static display, which can be stored on a web server and accessed as a PDF document. A better idea is to utilize the capacities built into the hyperref package, so that the document becomes an active PDF document. In particular, example expressions of the fragment now take the form of active links: a click of the mouse triggers the whole series of events described above, sending appropriate form data to the Perl script described above, and returning the results of the parse attempt as a PDF document. Additionally, the active form can contain text fields in which the user can enter arbitrarily constructed examples (compatible with the fragment), rather than simply selecting among the

examples listed in the fragment specification itself. To see the display of the fragment *knuth.pl*, visit http://grail.let.uu.nl/knuth.pdf.

Static libraries of fragments

Once it is possible to display a single fragment which allows active interaction with the GRAIL environment over the web, it is immediately possible to provide access to a library of fragments, each displayed as a PDF document prepared using the hyperref package and pdfLATEX. The interested reader can visit the fragments section of http://grail.let.uu.nl/tour.pdf for an example.

Dynamic libraries

Static libraries have their limitations. Most obviously, they depend on a webmaster to install fragments and make their displays available. For broader teaching, development, and research, it is preferable for web-based users to construct their own fragments and access displays of them, all in a way that allows the fragments to be tested and improved. Through the LWP.pm module, Perl makes it relatively simple to fetch remote files from hosts across the web. In brief, through interaction with text-fields in a web document, a remote GRAIL user can specify the URL of a fragment, specify a test expression and a test goal, and submit the test remotely. Resulting proof displays of the kind already seen are returned over the web. Alternatively, the user may request that a remote fragment be transformed into a an active LATEX document (as just discussed), which can be further used for teaching, development, and research.

FUTURE WORK

More complex forms of interaction involving fragment revision and editing directly via dynamic PDF documents would be desirable and are possible in principle. Also, the concept of a derivation itself is a dynamic notion: it would be nice to have an option to unfold derivations step by step in the typeset PDF document. We are currently experimenting with Stephan Lehmke's texpower bundle, which offers the required kind of functionality.

REFERENCES

Lambek, J. 1958, The mathematics of sentence structure. American Mathematical Monthly, **65**:154–170.

Lehmke, S. 2001, The TEXPower bundle. Currently available in a pre-alpha release from `http://ls1-www.cs.uni-dortmund.de/~lehmke/texpower/`.

Moortgat, M. 1997, Categorial type logics. Chapter 2, *Handbook of Logic and Language*. Elsevier/MIT Press, pp. 93–177.

Moot, R. 1998, Grail: an automated proof assistant for categorial grammar logics, *in* R. Backhouse, ed., 'Proceedings of the 1998 User Interfaces for Theorem Provers Conference', pp. 120–129.

Radhakrishnan, C.V. 1999, 'Pdfscreen.sty', Comprehensive TEX Archive Network. `macros/latex/contrib/supported/pdfscreen/`.

Rahtz, S. 2000, 'Hyperref.sty', Comprehensive TEX Archive Network. `macros/latex/contrib/supported/hyperref/`.

Tatsuta, M. 1997, 'Proof.sty', Comprehensive TEX Archive Network. `macros/latex/contrib/other/proof/proof.sty`.

Taylor, P. 1996, 'Prooftree.sty', Comprehensive TEX Archive Network. `macros/generic/proofs/taylor/prooftree.sty`.

◇ ◇ ◇

Poligraf: *from TEX to printing house*

Janusz Marian Nowacki*

abstract.
The macro package Poligraf was for the first time presented at the Polish TEX Users' Group meeting "BachoTeX'96". Users' suggestions and remarks have been taken into account leading to this new, completely re-worked version.

To my joy Poligraf has been received with significant interest. It turned out that a number of people use TEX for preparing publications for professional printing. Unfortunately, I must confess that the 1996 version had a number of shortcomings.

First, the full functionality of Poligraf was only available with the plain format, which I use daily. My lack of latex knowledge caused latex users, who form a majority, quite a number of problems.

Second, I attempted to solve too many issues at once. For example, I unnecessarily attempted to deal with assembling of print sheets. Now I know that there exist better and less error prone means to this end.

Third, the colour separation mechanism was too extensive and thus became too complicated. In the meantime, in parallel to my work the `cmyk-hax.tex` package has been under development by my colleagues from the BOP company in Gdańsk. I came to the conclusion that this is a far better solution than the one proposed in the first version of Poligraf.

The distribution

Currently the package consists of three files:[1]

- `poligraf.sty`—the main macros,
- `crops.pro`—a required header file to be used with the DVIPS program.
- `separate.pro`—an optional header file to be used with the DVIPS program[2].

*J.Nowacki@GUST.org.pl

[1]The package is to be found on the ftp server `ftp.gust.org.pl/GUST/contrib/poligraf.zip` and the "TEXLive 6" CD-ROM.

[2]Additionally, the `cmyk-hax.tex` package is required for colour separation.

In everyday practice the crops.pro file suffices

The majority of documents typeset with TEX or other DTP systems is printed using a desktop printer. On the other hand, if quality and size of the edition matter, more and more publications are being printed by professional offset print shops. For such purposes the document pages or print sheets should contain additional elements required by the printers. These are for example registration marks. crop marks, colour steps and colour bars.

In the current version of the POLIGRAF package this task is realised by the header file `crops.pro`, input by the DVIPS program[3].

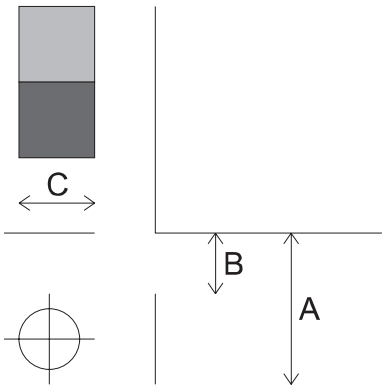
The `foo.dvi` file generated with any TEX format (PLAIN, LATEX2ϵ, $\mathcal{AMS}$TEX, etc.) is run through DVIPS,

```
dvips -h crops.pro foo.dvi
```

producing a `foo.ps` file with all the necessary elements.

The user may change the default values of the parameters. It suffices to change the values of some variables found in the first few lines of the file.

- ◇ `cropmarksize`—the size of the corner crop marks (A in the figure). The construct fits into a square hence only one dimension is necessary. The unit of measure is millimetres. The proposed default is 10 mm.
- ◇ `cropmarkdistance`—the distance of the crop marks from the page field (B). The unit of measure is millimetres. The proposed default distance is 3 mm.
- ◇ `barsize`—the size of the square fields (C), out of which the colour steps and colour bars are built. The unit of measure is millimetres. I propose the side of the squares to be 5 mm.



[3]The DVIPS program is distributed with its own `crop.pro` or `crop.lpr` header files. These are not satisfactory as they do not create all elements required by printers.

- ◇ `colorbars`—the switch controlling the range of control objects to be put onto the print sheet:
  - 0 : no colour strips, e.g. for print sheet assembling,
  - 1 : all colour strips,
  - 2 : only colour steps,
  - 3 : only colour bars.
- ◇ `mirror`—mirroring of whole print sheets including the colour strips and bars as well as registration and crop marks. No other mirroring methods should be used if this switch is on.
- ◇ `labeloff`—supresses the output of the TEX and POSTSCRIPT page numbers and the name of the separated colour.
- ◇ `xoffset` or `yoffset`—offset of the whole page including the marks and bars relative to the POSTSCRIPT co-ordinate origin.

Please note that the TEX source file should define the format of the print sheet. Problems arise if this is neglected.

TEX itself does not care about the sheet on which the publication is printed. Essential are the text width and height and the location of its upper left corner. The sheet format is used by DVIPS, which determines it using the information specified in the TEX source, for example:

```
\special{papersize=xmm,ymm}
```

If the sheet size is not specified in the source file, DVIPS uses the default value from the `config.ps` file, usually A4, which might not be what was expected.

Even with the sheet size specified in the source file, DVIPS may provide for surprising effects by "rounding" to a default format within an undefined tolerance. The following option

```
-t unknown
```

switches the formats off[4]. In this way one can make sure that the given sheet size will be used.

FROM WITHIN THE TEX FILE

DVIPS header files provide for independence of the TEX format, but not all users regard them as a convenient tool. A "real" TEX'er prefers to have full control over the publication from within the source file and this is why I wrote the new `poligraf.sty`. The `.sty` extension shows that now LATEX users, who constitute the majority of TEX users, are being supported.

To start using POLIGRAF the following invocation in the source file suffices:

```
\input poligraf.sty
```

[4]In some of the older DVIPS versions this option was incorrectly defined.

or in LATEX

```
\usepackage{poligraf}
```

This will cause the previously described `crops.pro` file to be inserted into the output file. Additionally the default values of its parameters may be overridden through the use of TEX directives of same names.

By using `poligraf.sty` one does not need to alter `crops.pro`, only the source file might need editing. Moreover, all of the parameters controlling the crop and registration marks as well as the colour bars and steps are saved in the source file, thus ensuring that the document will always have the same appearance.

Colour separation

The printing process requires that the publication be separated into the basic printing colours, i.e., cyan, magenta, yellow and black.

Normally this would be done by image setters. However, many users prepare the separations themselves. A reason for this could be the desire to judge the effects before the final image setting without incurring the cost. One could also produce uncomplicated separations using a laser printer and transparencies.

As has been said before, colour separations may be obtained with the proven macro package `cmyk-hax.tex` with its excellent choice of options. The package can be used not only for colour separation but also for manipulating the colours of individual objects.

Colour separations are achievable with the use of the following commands:

- ◇ `\Separate\CYAN`—separate cyan,
- ◇ `\Separate\MAGENTA`—separate magenta,
- ◇ `\Separate\YELLOW`—separate yellow,
- ◇ `\Separate\BLACK`—separate black,
- ◇ `\NoOverPrintBlack`—the standard behaviour of Poligraf is to overprint black paint on previously printed colours—this suppresses such behaviour.

The use of the `\Separate` command instructs TEX to input the `cmyk-hax.tex` package.

Colour separation at the command line

Colour separation may also be achieved with the help of the header file `separate.pro`. This file has been created with the help of the `cmyk-hax.tex` program. Several variables control its behaviour. For example, one can select the desired colour of the separation. The following command line

```
dvips -h separate.pro foo.dvi
```

produces the required POSTSCRIPT file. If the separation has to contain the print sheet's graphic elements the command

```
dvips -h separate.pro -h crops.pro foo.dvi
```

should be issued. The header files should be specified in the order shown.

Several file editing cycles are required to generate all colour separations. A more convenient solution is offered through the use of the four separation files: `cyan.pro`, `magenta.pro`, `yellow.pro` and `black.pro`. A batch file or a shell script might be created to generate all colour separations in a single action. The use of the header files `crops.pro` and `separate.pro` without the use of `poligraf.sty` allows one to process the `dvi` file directly. Sometimes this might be the way of choice with TEX formats I have not tested. Surprises might lurk there.

WHAT IS MISSING FROM THE NEW POLIGRAF?

The main reason for POLIGRAF to be re-written was the urge to have an easy to use program for as many TEX formats as possible. Several commands present in the previous version are missing from the new POLIGRAF:

- ◇ `\Language\Polski` and `\English`: were rarely used.
- ◇ `\Twoside`, `\Landscape`, `\LeftMargins`, `\TopMargins`: standard TEX solutions can be used instead.
- ◇ `\Hoffset`, `\Voffset`: have been replaced by `\xoffset` and `\yoffset`.
- ◇ `\ScrAngle`, `\ScrFrequency`, `\Rasterize`: the functionality is provided by the `cmyk-hax.tex` package.
- ◇ `\Hline`, `\Vline`, `\ShowGrid`, `\MargLines`: the same information is available from programs like ps-view and gs-view.
- ◇ `\beginLocalRaster`, `\endLocalRaster`: this is outside the scope of POLIGRAF.

ACKNOWLEDGEMENTS

I would like to thank cordially all the POLIGRAF users for bug reports and improvement suggestions. My thanks go also to Piotr Pianowski, Piotr Strzelczyk, Marcin Woliński and Staszek Wawrykiewicz for their effective help in writing the new version.

♢ ♢ ♢

Extending ExTEX

Simon Pepping

ABSTRACT. What can be done after the completion of ExTEX? I describe a dream, some results, and some further ideas.

KEYWORDS: ExTEX, DSSSL, file location, extension of ExTEX, primitives in ExTEX

SGML, DSSSL and TEX

Complex architectures have always appealed to me. It is no wonder then that the system SGML – DSSSL – typographical backend found in me a believer. Sebastian Rahtz's `jadetex` is a good implementation of TEX as the typographical backend. But ever since I had a look at it and at the way the `jade` DSSSL engine communicates with the backend, I have had this dream of a direct communication between the DSSSL engine and TEX, and of a direct implementation of flow objects in TEX.

One might think that this dream has been made obsolete by modern developments: SGML is out, XML is the new king. But I do not think things have really changed in the style specification area: The style language is now called XSL, its abstract page specification objects are now called formatting objects, but the idea has remained the same.

Obviously, I was not sufficiently skilled as a programmer to realize such a program. I also had the idea that with the current TEX program it would be impossible, and anyway I had little desire to work through the complicated canonical route to hack the TEX sources.

Therefore I looked forward to the release of the ongoing ExTEX project, so that I could try to realize my idea in a system with a more extensible structure. When I got the program in March, I started to try and understand its structure, and see where I should start to plug in my changes. That gave rise to some interesting conclusions.

TEX input without macros

When the `jade` program talks directly to the typographical backend, it makes calls to subroutines that start and end the flow objects, subroutines that register new values of characteristics, and subroutines that receive textual data. Those calls specify the layout in terms of abstract flow objects. From there the typographical backend should take it, and produce pages according to the abstract specification.

TEX, on the other hand, requires input in terms of its macro language. That has

been a great benefit. While the TEX program itself has proved hard to extend, the macro language has provided programmers/users with ample opportunity to write extensions.

The interprogram communication sketched above bypasses the macro language completely. As a consequence none of the extensions created in the 80s and 90s are available. When one looks at `jadetex`, one sees that it makes a good effort to pull in all major packages written for LATEX2ϵ. So, when we are not able to use those, we have a big problem. It is even worse: we have even bypassed plain TEX; indeed, we do not even have plain TEX's output routine, because that is specified through control sequences as well.

Karel Skoupy, the author of ExTEX, tells me that we do not even have a typesetting engine; we have no more than a library. It is the macro language that glues it all together into a typesetting engine. So my idea comes down to doing away with that glue!

That is where my plans have stalled. I will need some bright ideas to find a way forward.

Serving the files to ExTEX

When I had the ExTEX program, and took part in some discussions between its author and its users, it soon became clear that there was a problem in the way ExTEX finds the required files in the TEX distribution. Java, being a platform-independent language, has problems communicating with the environment. The chosen solution was to launch `kpsewhich` as a separate process to find the files for ExTEX.

While the `kpathsea` library has become a *de facto* standard, I am not ready to accept it as the only way to locate files in a TEX distribution, now and in the future. And therefore I do not think it a good idea to hard-code the use of `kpsewhich` into ExTEX. I prefer a separation between the typesetting engine and the TEX distribution, even though TEX has built-in file locating capabilities. I want a file locator architecture that is configurable by the distribution. The setup of Java's security mechanism, which is extremely configurable, with the possibility to plug in third party implementations of one or more functions, showed me how this could be achieved.

I constructed what I call the pluggable file locator architecture. ExTEX creates a `File Locator` object which defines the required file location functionality and its API. One or more implementations of that functionality may then be written, and added to ExTEX as modules. Start-up options determine which of the available implementations is actually used.

Basically, it works as follows. On the command line one has to tell ExTEX which file locator implementation one wishes to use. This can be done using the java property `nts.filelocatorclass`. Or it can be done in a configuration file, whose name should be communicated to the application using the java property `nts.filelocatorconfig`. An added bonus of a configuration file is that its path is communicated to the implementation, a feature which I use below in the `kpathsea` implementation. Extra arguments can be passed to the implementation. For more information, see the doc-

umentation in the java code itself (which is extractable with the javadoc tool). The idea is that the TEX distribution configures the command line to its needs, so that this is transparent to users.

Here is an example of the startup line of ExTEX using a file locator configuration file:

```
java -Dnts.fmt=latex \
 -Dnts.filelocatorconfig=/usr/share/TeX/bin/ntsfilelocator.cfg \
 Nts latex-file
```

The configuration file contains the following line:

```
nts.filelocator=<package>.<file locator implementation class>
```

Here is an example of a startup line directly providing the file locator implementation class. This class has a single constructor argument:

```
java -Dnts.filelocatorclass='<package>.kpsewhich \
                             /usr/share/TeX/bin/kpsewhich' \
 Nts tex-file
```

kpathsea

Of course, the *de facto* standard `kpathsea` was my first implementation. Since it is written in C, I used the Java Native Interface (JNI) to access it from the Java code. Hacking `kpathsea` turned out to be relatively easy. I took the code of `kpsewhich`, removed all code that is related to the command line options, added the functions that implement the Java interface, and I was almost done.

There was one complication: `kpathsea` is very much written for the situation where it is linked into an executable. Here the executable is `java`, which is of little help. So I have to supply artificially the name of another program, one in the `kpathsea` bin directory. I use `kpsewhich` for this purpose.

To my surprise, modifying in the `kpathsea` code the name of the executable from `java` to that of a `kpathsea` program:

```
kpse_set_program_name(NTSprogpath, NTSprogname);
```

at first did not work. It turns out that on `glibc` systems `kpathsea` does not use the two arguments of this function. The `glibc` library catches the program names and those are used by `kpathsea`. Therefore I have to reset the `glibc` program names:

```
program_invocation_name = NTSprogpath;
program_invocation_short_name = NTSprogname;
```

The file locator configuration file itself can be put in the `kpathsea` bin directory and used as a pseudo `kpathsea` program. When the first of the above ExTEX startup lines is used, the `File Locator` class communicates the path of the configuration file to the constructor of the implementation, and the latter uses it as the program path. This feature really embeds the file locator interface to ExTEX in the `kpathsea` setup of bootstrapping the distribution from the path of the executable.

TEX file server
As became apparent during discussions, Java is not good at communicating with the environment of the computer on which it runs. On the other hand, it is excellent in communicating with the networked world. So the idea presented itself to serve files over a network.[1] This was my second file locator implementation.

It was my goal to write a simple proof-of-concept. So I wrote a simple TEX file server using Java's ServerSocket class, and a simple TeXFSClient class as the file locator implementation that uses the TEX file server to get its files. The protocol is also simple:

Open TeX session The client sends a handshake to the server, to make sure the server can be found and is alive.

Open TeX file The client requests a file. It communicates the characteristics which are familiar from `kpathsea`: file name, format, must exist and program name.

This implementation basically demonstrates the idea. But it has some serious limitations:

- The session is not persistent, each file is requested in a new connection.
- In `kpathsea` it is not possible to change the name of the format, so once opened, the TEX file server serves files for only one format.

If this were to be turned into a serious tool, there is still a lot to be done. There are several possibilities to set up a more robust TEX file server. A good way to do it is over HTTP. It makes use of an established protocol, which is implemented in a number of excellent web server systems. To communicate TEX specific messages, one could use a CGI- and XML-based protocol, such as that recently developed by the Open Archives Initiative (see `http://www.openarchives.org/OAI/openarchivesprotocol.htm`).

Other extensions to ExTEX

LATEX2ε has some outstanding features which deserve to be separated from LATEX's document style features and made available to *all* TEX users:

NFSS Ever since the early 90s LATEX's font selection scheme has made it relatively easy to manage an ever growing font collection. Adding a new family of fonts is done in a systematic and transparent way.

graphicx/s providing the much needed integration of LATEX with graphics.

babel making (La)TEX multilingual.

LATEX has implemented a number of extended primitives, such as `\newcommand`, or the control sequences of the `ifthen` package, which make macro programming more robust and easier. These could be implemented as true primitives within ExTEX.

LATEX provides an interface to document classes, and for that purpose defines many LATEX primitives, such as `\@startsection`, the `counter` commands, the `option` commands. That interface can be realized in ExTEX itself, as a module. LATEX users can

[1] I understand that the file server idea has also been put forward within the NTS steering group.

use it, others can leave it alone.

These facilities (and LATEX itself) have been realized using TEX's extension mechanism, its macro language. This is an astonishing feat, in view of the fact that the macro language is hardly adequate as a programming language. Now that we have an extensible program, it seems a good idea to program these facilities as modules. As a programmer, I believe that the use of a programming language would make it much easier to achieve the desired logic.

Epilog

ExTEX has been awaited for a long time, and its realization is not entirely satisfactory: it is slow and resource hungry. But despite these drawbacks, we should appreciate that it is the first complete reimplementation of TEX ever made.

Its release provides us with a tool that we can play with, modify, and extend, more readily than we can do with TEX. At the very least, it is a sandbox on which new ideas can be tested. I hope the recount of my efforts provides inspiration to others to start working with the system.

♢ ♢ ♢

*Directions for the TEXLive system**

Fabrice Popineau

abstract. This paper is about the current status of the TEXLive software. The first part of the paper will address the structured description of its content and how the Windows [1] setup program can use it. The past experiments with the Windows installer have revealed that the problem was harder than expected. The new TEXLive 6 description files will allow a more effective way to use the setup program. Some further enhancements are even scheduled.
The second part of the paper will address a set of possible extensions to the Web2C/Kpathsea pair (read it as a call for code contributions!). Some aspects of its use were not foreseen when it was devised and it may be time for an enhancement.

keywords: TEXLive, Web2C

Introduction

TEX is a great piece of software, but it used to be not so easy to distribute. As time passed, people arranged so called *TEX distributions* for various platforms, which became easier and easier to install and use, even for people not really acquainted with TEX.

To name only a few, there was emTEX for DOS platforms, which came compiled, as usually people do not have development tools for this platform. There was also Web2C for Unix, which came in source form, because all Unixes have development tools. This concerns the programs. On the other side, the collection of TEX macros available around the Internet did not stop to inflate over the years.

All these distributions matured and Web2C turned out to be really portable so that its programs were made available on all major platforms. The most successful instantiation of Web2C as a standalone distribution was the great teTEX distribution assembled by Thomas Esser. At the same time, the TEX Directory Structure was standardized, and the layout of the support files for the programs almost frozen. It was time then to build the first multiplatform ready-to-use distribution: the TEXLive project was born.

*Thanks to Sebastian Rahtz for putting up the TEXLive project.

[1] Later in this paper, all the various Windows platforms will be designated by the generic Win32 acronym.

The 6th edition of the TEXLive system

The TEXLive system gathers almost all *free* TEX related software available from the CTAN repository. By this I mean all the programs directly related to TEX and its use (around 240 for Unix, a few more for Windows) and most of the free macro packages available on CTAN plus documentation. This is around 40000 files on a CD-ROM. So there is one big problem: how to manage such a huge pile of files?

We can split it in two parts. First, CTAN is great because it holds all TEX related files, it is managed, and you can find whatever you need on it if it does exist. But the files there are not directly usable: you need to unpack, compile, install at the right location on your site. Not something all people are ready to do or even not easy to do at all. Anyway, CTAN aims at being a repository for contributions to the TEX world, no more no less.

So in a sense the TEXLive system can be seen as a projection of CTAN onto some instance of the TDS with all files ready to use. This task has been almost completely automated for the macro packages by Sebastian Rahtz, who wrote a set of Perl scripts that can automatically convert any of these packages from their CTAN form into their usable form. That means that the archives are unpacked, the `.dtx` files are processed, documentation compiled, files moved to the right TEXLive location and so on.

The second problem is to find a way to present the user with those files in an understandable way. So we need to build a structured description of the files.

Structured description of the TEXLive system

The RDF format has been chosen for describing the components. RDF is an XML application that allows one to express sentences of the OAV (*Object – Attribute – Value*) kind. One can use it to claim that such url on the Internet has an attribute of such value. This model is powerful enough to express complex sentences, but we don't need the advanced features for the moment. However, relying on this language is a best bet for the future when we might need them. We have chosen a dedicated namespace for the so-called TPM (*TEX Package Management*) description files.

This edition introduces a new structured description of the various components. The current description is based on collections and packages. What is a collection and what is a package? A collection is intended to be a *collection of packages*. Each of these components has a common set of attributes which are given below for the `tex-basic` collection:

```
<!DOCTYPE rdf:RDF
  SYSTEM "../tpm.dtd">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:TPM="http://texlive.dante.de/">
   <rdf:Description about="http://texlive.dante">
      <TPM:Name>tex-basic</TPM:Name>
      <TPM:Date>2001/03/16</TPM:Date>
      <TPM:Version>1.0</TPM:Version>
      <TPM:Creator>rahtz</TPM:Creator>
      <TPM:Title>Essential programs and files
    </TPM:Title>
```

```
      <TPM:Description>
      These files are regarded as basic for any TeX system, covering
plain TeX macros, Computer Modern fonts, and configuration for common
drivers.
      </TPM:Description>
      <TPM:Flags default="yes"/>
      <TPM:RunFiles size="14957">
texmf/tex/generic/hyphen/hypht1.tex
...
texmf/tpm/collections/tex-basic.tpm
      </TPM:RunFiles>
      <TPM:BinFiles arch="i386-linux" size="4706207">
bin/i386-linux/texdoctk
...
bin/i386-linux/MakeTeXPK
</TPM:BinFiles>
      <TPM:BinFiles arch="win32" size="4194277">
</TPM:BinFiles>
      <TPM:Requires>
         <TPM:Package name="bibtex"/>
         <TPM:Package name="bluesky"/>
...
         <TPM:Package name="texlive"/>
         <TPM:Package name="tex-ps"/>
         <TPM:Package name="bibtex8bit"/>
      </TPM:Requires>
      <TPM:RequiredBy/>
   </rdf:Description>
</rdf:RDF>
```

Information about the components can be divided into three parts:

1. Information about the component: name, date, version, description, and so on. There is a special `Flags` tag, which is used to give directives to the setup program. For the moment, it is used to specify that some component should be selected by default, or that it is only remotely available.
2. Files lists, split into:

   bin files which are tagged by the target architecture given as an attribute and which specify all the files that should go into an architecture dependent directory (namely the programs);

   run files which refer to the files used by the TEX system and which are architecture independent, i.e. mostly macro files;

   source files which refer to the source files for the macro packages; the source files for the programs are not yet indexed this way;

   doc files which refer to the documentation files that come with the component;

   remote files which are used only for some Win32 support files that are not free or for which there is no space on the CD-ROM, hence that are only remotely available.

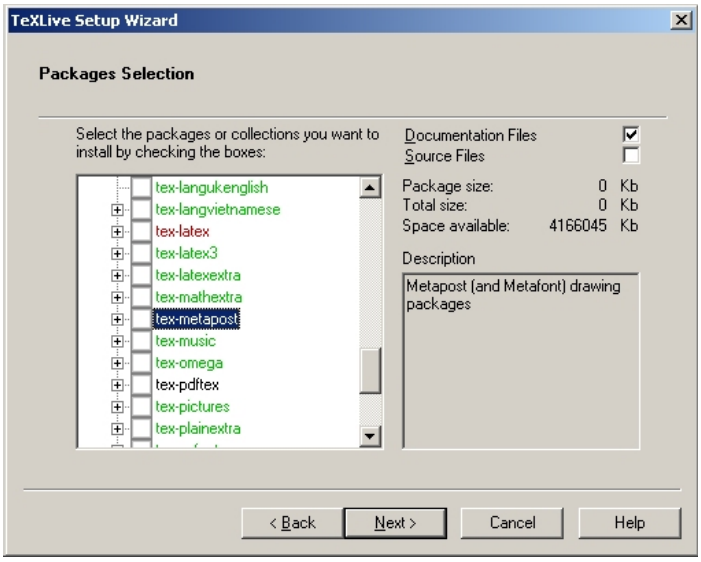


FIGURE 1: PACKAGE SELECTION PAGE OF THE TEXSETUP WIZARD.

3. Dependencies: the other required components needed to make this component work.

This information is sufficient to build a simple management system with the following features:

Installation install the files for the first time on the computer;

Maintenance allow the user to add, upgrade or remove components;

Removal remove all the files that have been installed.

All these operations should be done with consistency ensured. By this I mean that all the required packages are installed whenever you choose one, and that when you upgrade some package, all the required packages are upgraded if needed. Obviously, the database of TPM files needs to be sound by itself and should reflect the reality.

Enforcing consistency can be tricky. Dependencies define a graph between the components and the job of the management program when you ask it to install something is to find a minimum spanning tree over this graph that covers all the selected packages. Using such a system where consistency is enforced, it may happen that you will try and be denied to deselect some component: all dependencies may not be obvious to the user and it is rather intriguing to click on some checked box, and to fail to unckeck it! Maybe in this case the system should display the reason why the component is required. So dependencies and package bounds should be carefully designed.

Experiments have been done with the Windows TeXSetup program. The version in the current TEXLive system allows the user to install, to add and to upgrade packages. Figure 1 shows the screen when the user wants to add or upgrade some

of his packages. The black labels (only `tex-pdftex` on the figure) indicate that the component is installed and up to date, the red ones (`tex-latex` on the figure) that the component is out of date and the green ones that it is not yet installed (all the other packages on the figure).

How to go further in describing the TEXLive system?
The system is working properly at this point. However, not all issues have been adressed.

There is a small step that will be taken quickly[2]. Currently, packages as well as collections hold files. This should not be the case anymore. All files should be moved into packages, and collections shoud only hold a set of requirements. This will not prevent packages from requiring collections if that is needed. It will only ease the visualisation of dependencies.

In many cases where collections hold files, the files are binaries. So the binaries will also be split into packages corresponding more or less to the various parts in the program sources of the TEXLive system. So in this new description, the `tex-omega` collection will require some `omega-basic` collection and some `omega-lambda` collection, the `omega-basic` collection requiring the `omega-binaries`, `omega-base` and `omega-fonts` packages. Further, this will enable us to introduce a dependency between the `omega-binaries` package and the `kpathsea` package. You may wonder why it is needed. Under Windows, Kpathsea is distributed as a DLL as are some other parts of the system, so if you upgrade some program depending on Kpathsea, you potentially need to upgrade Kpathsea itself, and hence all other programs depending on it. But once this dependency is explicit, even upgrading the programs will be done in a safe way.

Another point that will probably be adressed is the removal of the division into doc-files, source-files and run-files. The problem is that at the moment you can choose to install each package with or without documentation and with or without sources. This has two flaws:

1. Each selected package is flagged as being installed, but nothing is remembered about its documentation or source files. The 'no-doc' flag is global, but the doc-files are defined per package. This is not so much of a problem as long as you can reinstall each package alone and choose to add its doc and source files if needed, but it is not very consistent because when you upgrade the package, you do not know if you should do it with or without doc and source files.
2. It is worse for some packages that hold only documentation, because they can be flagged as installed while only their `.tpm` file got installed (with the 'no-doc' option selected globally).

Given that documentation is very useful for many packages[3], the documentation files will be combined with the run files. On the other hand, the source files, which are usually useful only to a few people, will be described in a separate set of `.tpm` files, and there will be a separate collection (resp. package) for the source files of each collection

[2]Maybe even achieved by the time of the conference.
[3]And that the cost of disk-space has dramatically dropped!

(resp. package). This set of source files related collections and packages will form a separate tree from which users can select components. The setup program can have an option to automatically link any selected package to its source counterpart so that sources get installed easier.

Now, if we look further, we can wonder if a different, much bigger step should not be taken. We said that the TEXLive system is the projection of the CTAN content onto a ready-to-use axis. But CTAN already has its own set of description files, namely the Graham Williams Catalogue! Which incidentally is on the TEXLive CD-ROM. So we have two sets of description files. In fact, there is a third useful description of the components, which comes with the texdoctk program[4].

So we might think that three is two too many. One description should be enough. It is true that a merge between the Catalogue and the description files of the TEXLive system is highly desirable. But this is a huge work to undertake, and it could be done only by setting up a new procedure to submit files to CTAN. In this procedure package authors would be asked to build the XML description file for their package and also probably to follow very strict rules to package their contribution. Maybe this procedure can be automated by providing people with the right set of scripts, but this needs to be investigated.

Remaining problems about the setup system

The configuration files are not yet handled very nicely. The problem has been solved locally for the `language.dat` file. It has been split into several parts one line long, each one assigned to some collection specific to this language. When the user selects his collections, all the parts are assembled and the specific `language.dat` is build.

This process could be generalized to some other configuration files: `fmtutil.cnf`, map files in `pdftex.cfg` and `config.*` for dvips and gsftopk. However, doing this is hardcoded for the moment. It would be better to design and standardize some mechanism that would allow one to specify how the configuration files are built through rules and regular expressions.

Another remaining problem is about the setup programs. The Unix install-cd.sh script shell does not actually use the RDF files, but some preprocessed version of them. It also has no graphical user interface.

The Windows installer is a regular MFC dialog-box based application with a specific behaviour: a wizard in the Windows world. It relies heavily on platform specific features (dialog items like tooltips and tree control, Internet functions) so its code is not at all portable to the Unix world.

There is not yet any installer for the binaries on MacOS/X, and there is little chance that if one is written it could even be reused on another Unix system.

Reasoning in the context of a multiplatorm distribution, it would be nice to have a portable installer. However, requirements are very different between Unix and Windows. The other problem is that the installer needs to be standalone, and none of the tools that would allow one to quickly build a portable installler like Perl or Python, have the feature of building standalone programs. If this feature is ever introduced

[4] A nice Perl/Tk script that points you to the documentation inside a teTEX installation.

for all platforms, then this is clearly the way to go. But we can't expect that on every platform, users will have a complete Perl/Tk or wxPython installation, with the right GUI libraries, right version of the files and so on.

The other way of thinking, probably much more realistic, is to rely on a specific setup system for each platform. Sebastian Rahtz is willing to provide RPM packages for Debian and RedHat Linux users, and in fact he already did it as an experiment. Not all details have yet been sorted about – especially related to configuration files – but that's on the way for the next release.

Now that Microsoft has provided its own installer for all the versions of Windows, it would probably be better to switch to it[5]. It was not obvious that the first version of this system would be usable for the TEXLive system mainly because of the large number of files, but this Windows Installer system has quickly evolved and can't be ignored today.

So the common starting point will be the RDF description of the packages, and platform dependent versions will be built by scripts for each kind of setup system.

Other enhancements

We would like to get even more granularity. For instance, the "langgreek" collection has about five different Greek setups. Which one is best? In general, it is time we started *to cut down* on LATEX packages. Perhaps ConTEXt shows us the way forward.

The new package description allows us to define different distributions for different uses. Most people do not want the whole set of programs. We would like to define genuinely small TEX distributions, as small as possible. It could help to make TEX more popular. Setting up a pdfLATEX based distribution, without the METAFONT system but only Type1 fonts, for Western European countries, and a reasonable amount of LATEX packages takes less than 30Mb. Maybe it is still too much, but this is way less than the default setup of the previous TEXLive versions.

At the heart of the system

There are other parts of the system that could be enhanced to gain greater ease of use.

Unforeseen usage of Web2C

The `\write18` primitive was very successful among advanced TEX users. Using this primitive, you can make TEX execute commands while compiling your document, but you can also make these commands write their result into a file and `\immediate`ly read it! This is very interesting because it means you can make TEX communicate with other programs bi-directionally.

So what is the issue with an extensive usage of this command? The main problem is that the Kpathsea library has not been designed to be reentrant, and that none of the programs that are chained share any of the data structures used by Kpathsea. So the various hash-tables are built several times, which takes time, especially if you

[5]The latest experiments about permissions under Win2K would argue for this solution.

consider building the hash-table for the `ls-R` of the TEXLive system, which may hold all the files.

If we consider the Win32 platform, the situation is even worse under NT/Win2K and the NTFS filesystem because of the underlying Unicode filenames that need to be compared case-insensitively. It appears that the conversion towards uppercase names is a little bit slower under NTFS. Benchmarks on a PII-400Mhz machine show that building the hash table takes up between 2 s and 3 s without the conversion to uppercase names, and around 1 s more with it.

When I was told some people could call METAPOST hundreds of times from their pdfTEX run, each METAPOST run calling in turn pdfTEX, I quickly hacked Kpathsea to put the hash tables in shared memory, and save the initialisation time. The result was as expected. Processing the following file:

```
\input supp-pdf.tex

\pdfoutput=1

\immediate\openout10=fabrice.mp
\immediate\write10{beginfig(1);fill fullcircle scaled 4cm;endfig;end.}
\immediate\closeout10

\loop
  \immediate\write18{mpost fabrice.mp}
  \convertMPtoPDF{fabrice.1}{1}{1}\vfill\eject
  \ifnum\count0<100
\repeat

\end
```

with a debug version of kpathsea and the hacked version, time dropped from 2min 30s to 0min 30s. So there is indeed some potential gain for huge jobs there. Obviously, the final gain depends mostly on how much time is needed to do the typsetting itself.

However I think that, if we want to make the design of the whole system evolve, such a step will be needed. Kpathsea will need to act as a server. At what level needs to be discussed (probably a DCOM server on Windows, but read further about a plug and play TEX system).

Extending Kpathsea towards the Internet

Kpathsea is a nice library which has provided a common interface to *paths* (ways to access some file) across different platforms and architectures. It is not very strange that today people wonder if it could be extended towards the Internet. It would allow you to include files in your documents using urls. We could even define it as a new protocol and use it like this:

```
\RequirePackage{tds:graphics.sty}
...
\font\foo=http://www.font.net/foo/bar/x
```

In fact, there is nothing preventing us from doing so. Some experiments along this

line are on the way under Windows. The TeXSetup program required the ability to download files, so the basic functions to grab a url to a local file or to a string are there. It is not that hard to make kpathsea aware of urls as a kind of file. And it is another good test of the robustness of its API. The only problem is that downloading files takes time, and it is subject to failure because of dead connections and so on. So adding this feature should be done carefully:

- ◇ Whenever a file is downloaded, display some progression of the download so that users can decide to kill the job if it is stuck or too slow.
- ◇ Add a file caching system to avoid downloading the same file twice if it is used twice by the same job. This cache system should also be given a maximum amount of disk space to use and an LRU mechanism to decide which files to drop when it needs to reuse the space. One could also argue that this is the job of a proxy server, but not everybody can set up one. So we might be forced to implement a simple one.

These are the minimum requirements of such a feature.

Other possible extensions

Memory allocation should be made dynamic throughout the Web2C system. It is really annoying when you have to rebuild your format file because you have reached the current limits. And there is nothing in your document to specify that you used an enlarged TEX to compile it, so beware if you give the file to someone else. Moreover, almost half of the `texmf.cnf` file is dedicated to specifying those limits. Worse, not all the arrays used in those programs have been made extensible, but only the ones that are most likely to require it. However, every year, we need to add arrays to the list of extensible ones per users' requests.

Kpathsea should make it simpler to put files in the TDS tree. Currently, it is done by a very complex set of shell scripts. I know that some people advocate for shell scripts being easy to use and read but I'm not sure it is the vast majority. Being able to provide your own function to change the behaviour of Kpathsea, or to configure the way it will store files, why not. But in any case, Kpathsea should be complete and provide a decent, canonical and simple system to store the generated files. This is also a requirement for the point discussed further about a better integrated TEX system.

Core code considerations

Sharing and reusing code

Various routines are duplicated in several places, or implemented in different ways. These routines deal with very different things:

- ◇ Basic ones, like reading a line in a text file. If we want to support several platforms with different end-of-line conventions, then we should take care of the problem, preferably in only one place. It appears that for example, BibTEX does not use the same routine to perform this function as the one used by TEX. The TEX routine has been enhanced to support different eol conventions, but the BibTEX one has

not, so we eventually run into the buffer size problem with BibTeX when it is fed with a MacIntosh file.

- ◇ Higher level ones, like reading a DVI file. This is done several times by very different tools for different needs (xdvi may not have the same requirements as dvitype), but all in all, the feature is the same: being able to parse a DVI file and access all the objects in it. Maybe the DVIlib by Hirotsugu Kakugawa presented last year at TUG2000 could be a starting point.

Any project aiming at enhancing the current set of programs should have a close look at these redundancies, try to implement some kind of lowest common multiple set of features in terms of libraries, then make the programs use these libraries and remove the original code. It has been done once with great success: Kpathsea is the way to go. It can be done further. For example, there is the VFLib project, which could be used to let all the programs share several parts of code related to font handling.

Towards an integrated plug and play TEXsystem.
What is the aim of this discussion? We love TeX and we would like it to be more popular and shared by many more people. So what do we need to make it easier to use and install? Basic computer users are used to programs with a nice interface, preferably with lots of menus and buttons, they are not used to dealing with dozens of console mode programs. In this respect, any TeX system looks old.

So what do we have to do to turn TeX into something more appealing? We have for example XEmacs (or maybe Emacs, but I don't know much of the latest developments) which has a very nice design that could be made to embed TeX. How could it be done?

Currently, the way to run TeX under any Emacs flavor is with the AUC-TeX package. The standard way for Emacs to talk to external processes is through the console. But if we aim at fast communication between TeX and the embedding environment, we might want to remove the overhead of accessing the console (on both sides). XEmacs is already able to dynamically load a DLL.

On the Win32 platform, one is expected to build shared parts of programs as a DLL, much more than on Unix. The TeX engine, for example, is built as a DLL because there are no symbolic links under Win32. So the DLL is linked to various stubs (`tex.exe`, `latex.exe`, etc.) that just set the program name which will be used later to load the right format.

So if we want to go further in this direction, what we need is a TeX program independent of any console, that could be addressed by other programs which will dynamically open the `tex.dll` and communicate with it through its API. At the moment it is not really easy because TeX does not know how to report errors by any other way than writing to the log file or to the console.

The big picture would be to make XEmacs able to display DVI files into one of its buffers, which might actually not be so difficult. This could be done by turning some previewer into a DLL and linking XEmacs to it. Then what you might want is the ability for XEmacs to talk to the TeX DLL, and let the output of the TeX DLL feed the previewer. Jonathan Fine will present his *Instant Preview* system in these proceedings, which reuses the ability of Web2C engines to send their output unbuffered

through a socket. Jonathan's system splits the dvi output into smaller parts to achieve a better effect of instantaneously viewing the output of the text being typed in. If we go the way of sharing code in libraries and if some powerful DVI library is designed, then we could even remove the overhead of creating all those files, because the DVI library could allow any of its clients to access any page of any file currently known. A lock mechanism together with a copy on write mechanism could achieve even much better performance, and we would have a really nice typesetting system.

This is still fiction, but this kind of project can certainly be put up. So if you have students or man power to write code, please stand-up!

Conclusion

TEX used to be difficult to set up. Over the years, thanks to the many contributions of the TUG community, things have become smoother. However, the status is not yet up to the current standards of plug and play software. To go further in this direction requires us to rethink some of the core code.

I hope that by opening up the discussion on these topics I will have woken up some good will and that a new wave of contributions will appear: we need volunteers to make things move.

♢ ♢ ♢

DCpic, Commutative Diagrams in a (La)TEX Document

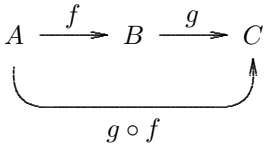
Pedro Quaresma*
CISUC
Departamento de Matemática, Universidade de Coimbra
3001-454 COIMBRA, PORTUGAL

abstract. DCpic is a package of TEX macros for graphing Commutative Diagrams in a (La)TEX or ConTEXt document. Its distinguishing features are: the use of PICTEX a powerful graphical engine, and a simple specification syntax. A commutative diagram is described in terms of its objects and its arrows. The objects are textual elements and the arrows can have various straight or curved forms.
We describe the syntax and semantics of the user's commands, and present many examples of their use.

keywords: Commutative Diagrams, (La)TEX, PICTEX

Introduction

Commutative Diagrams (Diagramas Comutativos, in Portuguese), are a kind of graphs which are widely used in Category Theory [4, 7, 9], not only as a concise and convenient notation but also for "arrow chasing", a powerful tool for mathematical thought. For example, the fact that in a Category we have arrow composition is easily expressed by the following commutative diagram.



The word commutative means that the result from going throught the path f plus g is equal to the result from going throught the path $g \circ f$. Most of the graphs used

*This work was partially supported by the Portuguese Ministry of Science and Technology (MCT), under the programme PRAXIS XXI.

in Category Theory are digraphs which we can specify in terms of its objects, and its arrows.

The (La)TEX approach to typesetting can be characterized as "logical design" [5, 6, 8], but commutative diagrams are pieces of "visual design", and that, in our opinion is the *piece de resistance* of commutative diagrams package implementation in (La)TEX. In a commutative diagrams package a user seeks the simplest notation, a logical notation, with the most powerful graphical engine possible, the visual part. The DCpic package, along with the package by John Reynolds [3, 10], has the simplest notation off all the commutative diagrams packages described in the Feruglio article [3]. In terms of graphical capabilities the PICTEX [12] package provides us with the best TEX-graphics engine, that is, without going to *Postscript* specials.

The DCpic package depends only of PICTEX and TEX, which means that you can use it in all formats that are based on these two. We have tested DCpic with LATEX, TEX plain, pdfLATEX, pdfTEX [11], and ConTEXt [8]; we are confident that it can be used under many other formats.

The present version (3.1) of DCpic package is available in CTAN and in the author's Web-page[1].

Constructing Commutative Diagrams

DCpic depends on PICTEX, thus you must include an apropriate command to load PICTEX and DCpic in your document, e.g. "`\usepackage{dcpic,pictex}`", in a LATEX document.

A commutative diagram in DCpic is a "picture" in PICTEX, in which we place our *objects* and *morphisms* (arrows). The user's commands in DCpic are: `begindc` and `enddc` which establishe the coordinate system where the objects will by placed; `obj`, the command which defines the place and the contents of each object; `mor`, and `cmor`, the commands which define the morphisms, linear and curved arrows, and its labels.

Now we will describe each of these commands in greater detail.

The Diagram Environment

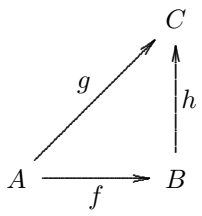
The command `begindc`, establishes a Cartesian coordinate system with 1pt units,

```
\begindc[<magnification factor>] ... \enddc
```

such a small unit gives us a good control over the placement of the graphical objects, but in most of the diagrams not involving curved arrows such a "fine grain" is not desirable, so the optional argument specifies a magnifying factor $m \in \mathbb{N}$, with a default value of 30. The advantage of this decision is twofold: we can define the "grain" of the diagram, and we can adjust the size of the diagram to the available space.

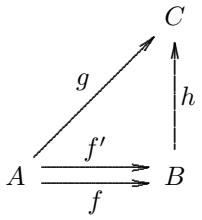
- ◇ a "course grain" diagram is specified almost as a table, with the numbers giving us the lines and the columns were the objects will be placed, the following diagram has the default magnification factor:

[1] http://www.mat.uc.pt/~pedro/LaTeX/



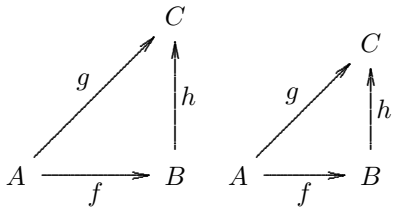

```
\begindc
\obj(1,1){$A$}
\obj(3,1){$B$}
\obj(3,3){$C$}
\mor(1,1)(3,1){$f$}[\atright,\solidarrow]
\mor(1,1)(3,3){$g$}
\mor(3,1)(3,3){$h$}[\atright,\solidarrow]
\enddc
```

◇ a "fine grain" diagram is a bit harder to design but it gives us a better control over the objects placement, the following diagram has a magnification factor of three, this gives us the capability of drawing the arrows f and f' very close together:




```
\begindc[3]
\obj(10,10){$A$}
\obj(30,10){$B$}
\obj(30,30){$C$}
\mor(10,9)(30,9){$f$}[\atright,\solidarrow]
\mor(10,11)(30,11){$f^\prime$}
\mor(10,10)(30,30){$g$}
\mor(30,10)(30,30){$h$}[\atright,\solidarrow]
\enddc
```

◇ the magnification factor gives us the capability of adapting the size of the diagram to the available space, without having to redesign the diagram, for example the specification of the next two diagrams differs only in the magnification factor: 30 for the first; and 25 for the second.



Note that the magnification factor does not interfere with the size of the objects, but only with the size of the diagram as a whole.

After establishing our "drawing board" we can begin placing our "objects" on it, we have three commands to do so, the `obj`, `mor`, and `cmor`, for objects, morphisms, and "curved" morphisms respectively.

Objects

Each object has a place and a content

`\obj(<x>,<y>){<contents>}`

the x and y, integer values, will be multiplied by the magnifying factor. The *contents* will be put in the centre of an "hbox" expanding to both sides of $(m \times x, m \times y)$.

Linear Arrows
Each linear arrow will have as mandatory arguments two pairs of coordinates, the beginning and the ending points, and a label,

`\mor(<x1>,<y1>)(<x2>,<y2>)[<d1>,<d2>]{<label>}[<label placement>,<arrow type>]`

the other arguments are opcional. The two pairs of coordinates should coincide with the coordinates of two objects in the diagram, but no verification of this fact is made. The line connecting the two points is constructed in the following way: the beginning is given by a point 10pt away from the point $(m \times x_1, m \times y_1)$, likewise the end point is 10 points away from $(m \times x_2, m \times y_2)$. If the "arrow type" specifies that, a tail, and a pointer (arrow) will be added. The label is placed in a point (x_l, y_l) at a distance of 10 points from the middle point of the arrow, the position of the "hbox" is dependent of the angle and the direction of the arrow, if the arrow is horizontal the "hbox" will be centred in (x_l, y_l), if the arrow is vertical the "hbox" will be left, or right, justified in (x_l, y_l), and similarly for the other cases. In all cases the position of the "hbox" is such that the contents of it will not interfere with the line.

The distance from the point $(m \times x_1, m \times y_1)$ to the actual beginning of the arrow may be modified by the user with the specification of d_1, the same thing happens for the arrow actual ending in which case the user-value will be d_2. The specification of d_1 and d_2 is optional.

The placement of the label, to the left (default value), or to the right, and the type of the arrow: a solid arrow (default value), a dashed arrow, a line, an injection arrow, or an application arrow, are the last optional arguments of this command.

Quadratic Arrows
The command that draws curved lines in DCpic uses the `setquadratic` command of PICTEX, this will imply a quadratic curve specified by an odd-number of points,

`\cmor(<list of points>)␣<arrow direction>(<x>,<y>){<label>}[<arrow type>]`

the space after the list of points is mandatory. After drawing the curved line we must put the tip of the arrow on it, at present it is only possible to choose from: up, down, left, or right pointing arrow, and we must explicitly specify what type we want. The next thing to draw it is the arrow label, the placement of that label is determined by the x, and y values which give us the coordinates, after being magnified, of the centre of the "hbox" that will contain the label itself.

The arrow type is an optional argument, its default value is a solid arrow, the other possible values are a dashed arrow and a line, in this last case the arrow tip is omitted. The arrow type values are a subset of those of the `mor` command.

A rectangular curve with rounded corners is easy to specify and should cater for most needs, with this in mind we give the following tip to the user: to specify a rectangular, with rounded corners, curve we choose the points which give us the *expanded chess-horse movement*, that is, (x, y), $(x \pm 4, y \mp 1)$, $(x \mp 1, y \pm 4)$, or (x, y),$(x \pm 1, y \mp 4)$, $(x \mp 4, y \pm 1)$, those sets of points will give us the four corners of the rectangle; to form the whole line it is only necessary to add an odd number of points joining the two (or more) corners.

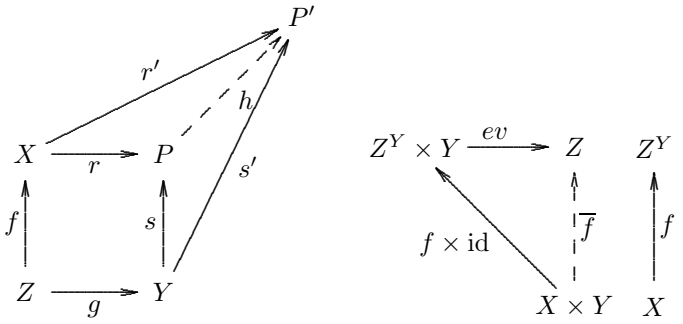
Examples

We now present some examples that give an idea of the DCpic package capabilities. We will present here the diagrams, and in the appendix the code which produced such diagrams.

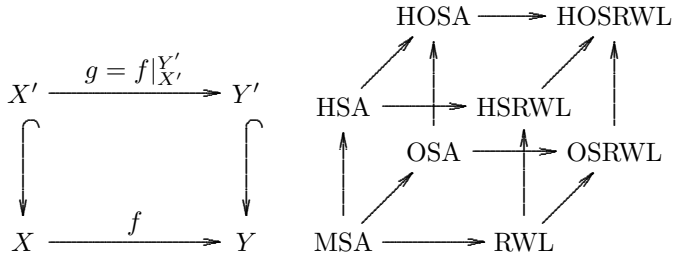
The Easy Ones

The diagrams presented in this section are very easy to specify in the DCpic syntax, just a couple of objects and the arrows joining them.

Push-out and Exponentials:



Function Restriction and the CafeOBJ *Cube [2]*

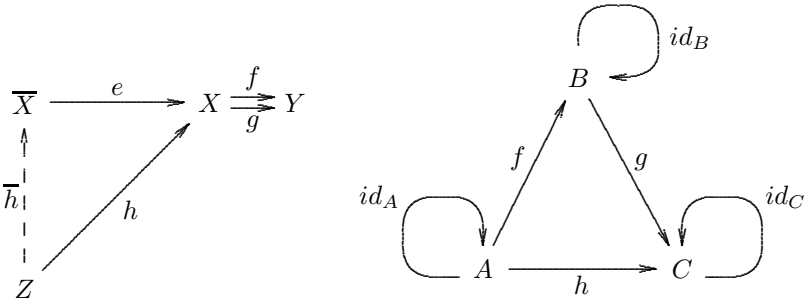


The Not so Easy

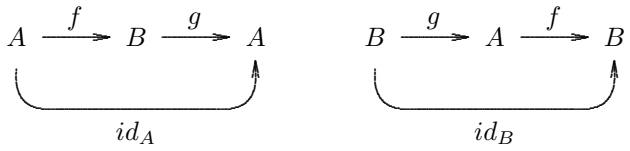
The diagrams presented in this section are a bit harder to specify. We have curved arrows, and also double arrows. The construction of the former was already explained. The double arrow (and triple, and ...) is made with two distinct arrows drawn close to each other in a diagram with a very "fine grain", that is, using a magnifying factor of just 2 or 3.

All the diagrams were made completely within DCpic.

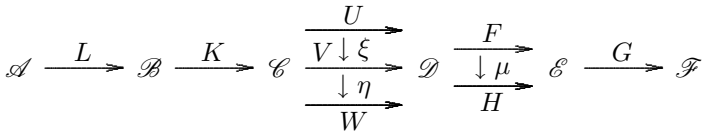
Equaliser, and a 3-Category:



Isomorfisms:



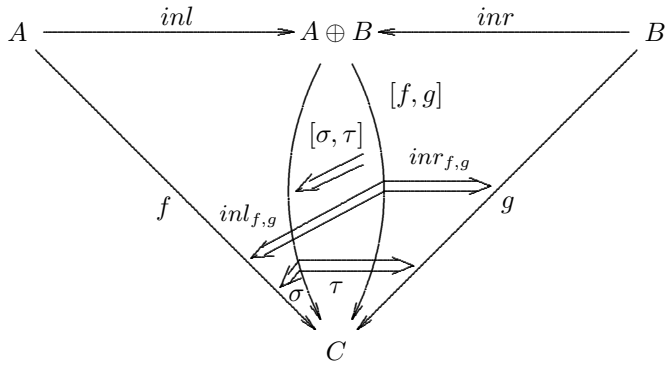
Godement's "five" rules [4]:



The others ...

It was already stated that some kinds of arrows are not supported in DCpic, e.g., $\Rightarrow$, but we can put a PICTEX command inside a DCpic diagram, so we can produce a diagram like the one that we will show now. Its complete specification within DCpic is not possible, at least for the moment.

Lax coproduct [1]



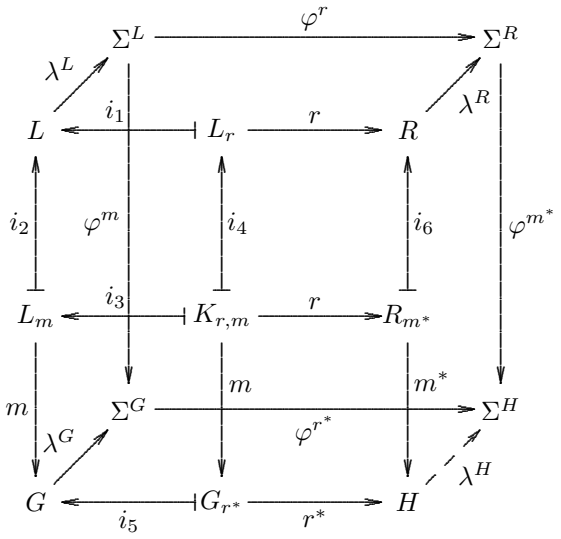
DCpic compared

If one took the Feruglio article [3] about typesetting commutative diagrams in (La)TEX we can say that:

- ◇ the graphical capabilities of DCpic are among the best. Excluding packages which use Postscript specials the DCpic package is the best among available packages.
- ◇ the specification syntax is one of the simplest, the package by John Reynolds has a very similar syntax.

We did not try to take any measure of computational performance.

The following diagram is one of the test-diagrams used by Feruglio, as we can see DCpic performs very well, drawing the complete diagram based on a very simple specification.



Conclusions

We think that DCpic performs well in the "commutative diagrams arena", it is easy to use, with its commands we can produce the most usual types of commutative diagrams, and if we accept the use of PICTEX commands, we are capable of producing any kind of diagram. It is also a (La)TEX-only package, that is, the file produced by DCpic does not contain any Postscript special, neither any special font, which in terms of portability is an advantage.

The author and his colleagues in the Mathematics Department of Coimbra University have been using the (now) old version (2.1) of DCpic for some time with much success, some of the missing capabilities of the older version were incorporated in the new version (3.1), and the missing capabilities of the new version will be taken care in future versions.

REFERENCES

[1] S. Abramsky, Dov Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*, volume 1 of *Oxford Science Publications*. Claredon Press, Oxford, 1992.

[2] Rãzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST series in Computing*. World Scientific, 1998.

[3] Gabriel Valiente Feruglio. Typesetting commutative diagrams. *TUGboat*, 15(4):466–484, 1994.

[4] Horst Herrlich and George Strecker. *Category Theory*. Allyn and Bacon Inc., 1973.

[5] Donald E. Knuth. *The TeXbook*. Addison-Wesley Publishing Company, Reading,Massachusetts, 1986.

[6] Leslie Lamport. LATEX*: A Document Preparation System*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1994.

[7] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971.

[8] Ton Otten and Hans Hagen. *ConTEXt an excursion*. Pragma ADE, Hasselt, 1999.

[9] Benjamin Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. The MIT Press, London, England, 1998.

[10] John Reynolds. *User's Manual for Diagram Macros*. http://www.cs.cmu.edu/~jcr/, 1987. `diagmac.doc`.

[11] Hàn Thế Thành, Sebastian Rahtz, and Hans Hagen. *The pdfTeX manual*, 1999.

[12] Michael Wichura. *The PICTEX Manual*. M. Pfeffer & Co., New York, 1987.

Appendix: The DCpic Specifications

Push-out:

```
\begindc[26]
\obj(1,1){$Z$}
\obj(1,3){$X$}
\obj(3,1){$Y$}
\obj(3,3){$P$}
\obj(5,5){$P^\prime$}
\mor(1,1)(1,3){$f$}
\mor(1,1)(3,1){$g$}[\atright,\solidarrow]
\mor(1,3)(3,3){$r$}[\atright,\solidarrow]
\mor(3,1)(3,3){$s$}
\mor(1,3)(5,5){$r^\prime$}
\mor(3,1)(5,5){$s^\prime$}[\atright,\solidarrow]
\mor(3,3)(5,5){$h$}[\atright,\dasharrow]
\enddc
```

Exponentials:

```
\begindc
\obj(1,3){$Z^Y\times Y$}
\obj(3,3){$Z$}
\obj(3,1){$X\times{}Y$}
\obj(4,1){$X$}
\obj(4,3){$Z^Y$}
\mor(1,3)(3,3)[20,10]{$ev$}
\mor(3,1)(1,3){$f\times{}\mathrm{id}$}
\mor(3,1)(3,3){$\overline{f}$}[\atright,\dasharrow]
\mor(4,1)(4,3){$f$}[\atright,\solidarrow]
\enddc
```

Function Restriction:

```
\begindc[28]
\obj(1,1){$X$}
\obj(1,3){$X^\prime$}
\obj(3,1){$Y$}
\obj(3,3){$Y^\prime$}
\mor(1,1)(3,1){$f$}
\mor(1,3)(1,1){}[\atright,\injectionarrow]
\mor(3,3)(3,1){}[\atright,\injectionarrow]
\mor(1,3)(3,3){$g=f|^{Y^\prime}_{X^\prime}$}
\enddc
```

CafeOBJ *Cube:*

```
\begindc[17]
\obj(1,1){MSA}
\obj(5,1){RWL}
\obj(3,3){OSA}
\obj(7,3){OSRWL}
\obj(1,4){HSA}
\obj(5,4){HSRWL}
\obj(3,6){HOSA}
\obj(7,6){HOSRWL}
\mor(1,1)(5,1)[15,15]{}
\mor(1,1)(1,4){}
\mor(1,1)(3,3){}
\mor(5,1)(5,4){}
\mor(5,1)(7,3){}
\mor(3,3)(3,6){}
\mor(3,3)(7,3)[15,22]{}
\mor(7,3)(7,6){}
\mor(1,4)(5,4)[15,22]{}
\mor(1,4)(3,6){}
\mor(3,6)(7,6)[17,26]{}
\mor(5,4)(7,6){}
\enddc
```

Equaliser:

```
\begindc[2]
\obj(1,1){$Z$}
\obj(1,36){$\overline{ X}$}
\obj(36,36){$X$}
\obj(52,36){$Y$}
```

```
\mor(1,1)(1,36){$\overline{ h}$}[\atleft,\dasharrow]
\mor(1,1)(36,36){$h$}[\atright,\solidarrow]
\mor(1,36)(36,36){$e$}
\mor(36,37)(52,37)[8,8]{$f$}
\mor(36,35)(52,35)[8,8]{$g$}[\atright,\solidarrow]
\enddc
```

A 3-Category:

```
\begindc[3]
\obj(14,11){$A$}
\obj(39,11){$C$}
\obj(26,35){$B$}
\mor(14,11)(39,11){$h$}[\atright,\solidarrow]
\mor(14,11)(26,35){$f$}
\mor(26,35)(39,11){$g$}
\cmor((11,10)(10,10)(9,10)(5,11)(4,15)(5,19)(9,20)(13,19)(14,15))
 \pdown(1,20){$id_A$}
\cmor((42,10)(43,10)(44,10)(48,11)(49,15)(48,19)(44,20)(40,19)(39,15))
 \pdown(52,20){$id_C$}
\cmor((26,39)(27,43)(31,44)(35,43)(36,39)(35,36)(31,35)) \pleft(40,40){$id_B$}
\enddc
```

Isomorfisms:

```
\begindc[3]
\obj(10,15){$A$}
\obj(40,15){$A$}
\obj(25,15){$B$}
\mor(10,15)(25,15){$f$}
\mor(25,15)(40,15){$g$}
\cmor((10,11)(11,7)(15,6)(25,6)(35,6)(39,7)(40,11)) \pup(25,3){$id_A$}
\obj(55,15){$B$}
\obj(85,15){$B$}
\obj(70,15){$A$}
\mor(55,15)(70,15){$g$}
\mor(70,15)(85,15){$f$}
\cmor((55,11)(56,7)(60,6)(70,6)(80,6)(84,7)(85,11)) \pup(70,3){$id_B$}
\enddc
```

Godement's "five" rules:

```
\begindc[7]
\obj(12,10){$\mathcal{A}$}
\obj(19,10){$\mathcal{B}$}
\obj(26,10){$\mathcal{C}$}
\obj(34,10){$\mathcal{D}$}
\obj(41,10){$\mathcal{E}$}
\obj(48,10){$\mathcal{F}$}
\mor(12,10)(19,10){$L$}
\mor(19,10)(26,10){$K$}
\mor(26,10)(34,10){$V\qquad\ $}
\mor(26,12)(34,12){$U$}
\mor(26,12)(34,12){$\downarrow\xi$}[\atright,\solidarrow]
\mor(26,8)(34,8){$\downarrow\eta$}
\mor(26,8)(34,8){$W$}[\atright,\solidarrow]
\mor(34,11)(41,11){$F$}
\mor(34,9)(41,9){$\downarrow\mu$}
\mor(34,9)(41,9){$H$}[\atright,\solidarrow]
```

```
\mor(41,10)(48,10){$G$}
\enddc
```

Lax coproduct: Guess how.

DCpic and the others:

```
\begindc[35]
\obj(1,1){$G$}
\obj(3,1){$G_{r^*}$}
\obj(5,1){$H$}
\obj(2,2){$\Sigma^G$}
\obj(6,2){$\Sigma^H$}
\obj(1,3){$L_m$}
\obj(3,3){$K_{r,m}$}
\obj(5,3){$R_{m^*}$}
\obj(1,5){$L$}
\obj(3,5){$L_r$}
\obj(5,5){$R$}
\obj(2,6){$\Sigma^L$}
\obj(6,6){$\Sigma^R$}
\mor(1,1)(2,2){$\lambda^G$}
\mor(3,1)(1,1){$i_5$}[\atleft,\aplicationarrow]
\mor(3,1)(5,1){$r^*$}[\atright,\solidarrow]
\mor(5,1)(6,2){$\lambda^H$}[\atright,\dasharrow]
\mor(2,2)(6,2){$\varphi^{r^*}$}[\atright,\solidarrow]
\mor(1,3)(1,1){$m$}[\atright,\solidarrow]
\mor(1,3)(1,5){$i_2$}[\atleft,\aplicationarrow]
\mor(3,3)(1,3)[14,10]{$i_3\quad$}[\atright,\aplicationarrow]
\mor(3,3)(5,3)[14,10]{$r$}
\mor(3,3)(3,5){$i_4$}[\atright,\aplicationarrow]
\mor(3,3)(3,1){$\stackrel{\displaystyle m}{\barraB}$}
\mor(5,3)(5,5){$i_6$}[\atright,\aplicationarrow]
\mor(5,3)(5,1){$\stackrel{\displaystyle m^*}{\barraA}$}
\mor(1,5)(2,6){$\lambda^L$}
\mor(3,5)(1,5){$i_1\quad$}[\atright,\aplicationarrow]
\mor(3,5)(5,5){$r$}
\mor(5,5)(6,6){$\lambda^R$}[\atright,\solidarrow]
\mor(2,6)(2,2){$\varphi^m$}[\atright,\solidarrow]
\mor(2,6)(6,6){$\varphi^r$}
\mor(6,6)(6,2){$\varphi^{m^*}$}
\enddc
```

♢ ♢ ♢

Using pdfTEX in a PDF-based imposition tool

Martin Schröder
Crüsemannallee 3, 28213 Bremen, Germany*

pdfTEX has been used successfully to build an industrial-strength PDF-based imposition tool. This paper/talk describes the pitfalls we encountered and the lessons learned.

*Net address: martin@oneiros.de, URL: http://www.oneiros.de

ASCII-Cyrillic and its converter email-ru.tex

by Laurent Siebenmann

A new faithful ASCII representation for Russian called ASCII-Cyrillic is presented here, one which permits accurate typing and reading of Russian where no Russian keyboard or font is available -- as often occurs outside of Russia.

ASCII-Cyrillic serves the Russian and Ukrainian languages in parallel. This article initially discusses Russian; but, further along, come the modifications needed to adapt to the Ukrainian alphabet.

TeX is mother to ASCII-Cyrillic inasmuch as its converter "email-ru.tex" is a program in TeX language which is interpreted by TeX. On the other hand, ASCII-Cyrillic is designed to be useful beyond TeX. Indeed a current project is to make it available on Internet so that the vast public ignorant of TeX can exploit it. This provisional Internet bias has led to the use of HTML for this article, bitmaps and all.

Contents

- **Overview**
- **Crash course on Russian ASCII-Cyrillic**
- **Ukrainian ASCII-Cyrillic**
- **Vital statistics for ASCII-Cyrillic**

Below is a photo of a fragment of Russian email sent from France to Russia. Ideally, it would be typed as 8-bit text using a Russian keyboard and screen font, and then assigned a suitable MIME type identifying the font encoding. To the email system, the message contents would be a sequence of "octets" or "bytes" (each 8 zeros or ones), where each octet corresponds to a character according to the font encoding. The

receiving email system and email reader are expected to recognize the encoding and provide for Cyrillic display and printing. This system works well provided there is diligent support of it from one end of the email trajectory to the other. The transcoding provided by "email-ru.tex" can be part of this support.

> На обратном пути Michele объяснила мне, как
> делать пересадку на метро. Мы с ней проехали
> большую часть пути вместе. Она вышла на
> остановке после того, как мы пересели на мою
> линию. Пользоваться метро №13A,
> действительно, очень просто -- гораздо проще,
> чем в Москве. Когда я это поняла, то сразу
> успокоилась. Сейчас всё в порядке. Я могу
> пользоваться метро, и уже не боюсь ходить в
> Париже.

(The GIF photo image you see here is widely readable, but at least 10 times as bulky as 8-bit text, and somewhat hazy too.)

Unfortunately, quite a few things can go wrong in MIME-tagged 8-bit Cyrillic email, particularly when either sender or recipient is outside the countries using a Cyrillic alphabet:

-- there is a frequent need to re-encode for another computer operating system, and when the targeted encoding does not contain all the characters used, defects result. Worse, if at any stage wrong assumptions are made about an encoding, severe and irreparable damage usually ensues.

-- outside the Cyrillic world, Cyrillic keyboards are rarissime, and Cyrillic screen fonts often have to be hunted down and installed by an expert.

To circumvent such difficulties Russian speakers often use an ad hoc 7-bit ASCII transliteration of Russian (or even switch to English) and then rely on ASCII's universal portability. ASCII, the American Standard for Computer Information Interchange of 1963, long predates Internet and the MIME protocols.

ASCII-Cyrillic is a new faithful ASCII transcription of Russian that transforms this "last resort" ASCII approach into something quite general and reliable.

For example, the email fragment illustrated above was first typed as the ASCII-Cyrillic text below, using a Latin (French) keyboard, and then converted to the above Cyrillic form by the utility "email-ru.tex". For good measure, both forms were then emailed.

```
Na obratnom puti !Michele obq'asnila mne, kak
delath peresadku na metro. My s nej proexali
bolhwu'u 'casth puti vmeste. Ona vywla na
ostanovke posle togo, kak my pereseli na mo'u
lini'u. Polhzovaths'a metro 'N13!A,
dejstvitelhno, o'cenh prosto -- gorazdo pro'we,
'cem v Moskve. Kogda 'a 'eto pon'ala, to srazu
uspokoilash. Sej'cas vs'o v por'adke. 'A mogu
polhzovaths'a metro, i u'ze ne bo'ush xodith v
Pari'ze.
```

Inversely, for email from Russia to France, the keyboarding would be Cyrillic and "email-ru.tex" would convert from 8-bit text to ASCII-Cyrillic. Again, for good measure, both versions would be sent.

ASCII-Cyrillic is designed to be both typeable and readable on every computer worldwide: Well chosen ASCII letters stand for most Russian letters. To distinguish the remaining handful of Russian letters, a prefixed accent ' is used. Further, to introduce English words, the exclamation mark ! appears. The rules are so simple that, hopefully, ASCII-Cyrillic typing and reading of Russian can be learned in an hour, and perfected in a week.

An essential technical fact to retain is that all the characters used by ASCII-Cyrillic are 7-bit (i.e. the 8th bit of the corresponding octet is zero), and each character has a reasonably well-defined meaning and shape governed by the universally used ASCII standard. It is a key fact that all 8-bit Cyrillic text encodings include and respect the ASCII standard where 7-bit characters are concerned.

In 7-bit ASCII-Cyrillic form, Russian prose is about 5 percent bulkier than when 8-bit encoded. Thus, typing speed for ASCII-Cyrillic on any computer keyboard can approach that for a Cyrillic keyboard.

The difference of 5 percent in bulk drops to about 1 or 2 percent when the common "gzip" compression is applied to both. Thus, there is virtually no penalty for storing Cyrillic text files in ASCII-Cyrillic form.

As "email-ru.tex" converts both to and from ASCII-Cyrillic, one can convert in two steps between any two common 8-bit Cyrillic encodings. Further, new or "variant", 8-bit encodings can be quickly introduced "on-the-fly" by specifying an "encoding vector". Additionally, the Cyrillic UTF8 unicode will soon be supported.

ASCII-Cyrillic is a cousin of existing transcriptions of Russian which differ in using the concept of ligature -- i.e. they use two or more English letters for certain Russian letters. The utility "email-ru.tex" also converts Russian to one such ligature-based transcription system established by the the USA Library of Congress:

```
Na obratnom puti Michele ob'jasnila mne, kak
delat' peresadku na metro. My s nej proexali
bol'shuju chast' puti vmeste. Ona vyshla na
ostanovke posle togo, kak my pereseli na moju
liniju. Pol'zovat'sja metro No13A,
dejstvitel'no, ochen' prosto -- gorazdo proshche,
chem  v Moskve. Kogda ja eto ponjala, to srazu
uspokoilas'. Sejchas vse v porjadke. Ja mogu
pol'zovat'sja metro, i uzhe ne bojus' xodit' v
Parizhe.
```

Nota bene:- Accurate reconversion of existing ligature-based transcriptions back to 8-bit format always requires a good deal of human intervention.

Although not more readable, the ASCII-Cyrillic representation has the advantage that, for machines as well as men, it is completely unambiguous as well as easily readable. The "email-ru.tex" utility does the translation both ways without human intervention, and the conversion (8-bit) ==> (7-bit) ==> (8-bit) gives back exactly the original 8-bit Russian text. (One minor oddity to remember: terminal spaces on all lines are ignored.)

Thus, by ASCII-Cyrillic encoding a Russian text file, one can archive and transfer it conveniently and safely, even by email.

Beginner's operating instructions

To use "email-ru.tex" as a converter:

- Put a copy of the file to convert, alongside of "email-ru.tex" and give it the name "IN.txt".

- Process "email-ru.tex" (not "IN.txt") with Plain TeX. The usual command line is: `tex email-ru.tex`

- Follow the instructions then offered on screen by "email-ru.tex".

A batch mode will soon be available.

Use of ASCII-Cyrillic with TeX

ASCII-Cyrillic could be made completely independent of TeX through using, to build its converter, some other portable language (C, Java, ...). On the other hand, the TeX community, with its keen appreciation of ASCII text as a stable portable medium, will probably always be "home ground" for ASCII-Cyrillic. Thus, it is unfortunate that, for lack of time, this author has not so far created macro packages offering optimal integration of ASCII-Cyrillic into Cyrillic (La)TeX typesetting. Anyone taking up this challenge is invited to contact the author -- who would like to use such macros for definitive ASCII-Cyrillic documentation!

In the interim, one has a simple modus vivendi with essentially all TeX formats having 8-bit Cyrillic capablility -- one which requires no new macros at all! Namely, convert from ASCII-Cyrillic text to 8-bit Cyrillic text (with embedded TeX formatting), and then compose with TeX. (As will be explained, the TeX formatting commands are largely unchanged when expressed in ASCII-Cyrillic.) The converter "email-ru.tex" then serves as a preprocessor to TeX. One way to get good value from this approach is to break your TeX typescript into files that are either purely ASCII or else mostly Cyrillic. Only the latter sort undergo conversion. The two sorts of file can then be merged using TeX's `\input` command or LaTeX's `\include`.

Snag Warning

A few important TeX implementations, notably C TeX under unix, and a majority of implementations for the Macintosh OS, are currently unable to `\write` true octets > 127 --- as "email-ru.tex" requires in converting from ASCII-Cyrillic to 8-bit Cyrillic text. (This problem does not impact the conversion from 8-bit Cyrillic text to ASCII-Cyrillic.)

To solve this problem when it arises, the ASCII-Cyrillic package will rely on a tiny autonomous and portable utility "Kto8" that converts into genuine 8-bit text any text file which the few troublesome TeX installations may output.

The sign that you need to apply this utility is the appearance of many pairs ^^ of hat characters in the output of "email-ru.tex".

Ready-to-run binary versions of "Kto8" will progressively be provided for the linux, unix, Macintosh, and Windows operating systems. The most current distribution of "Kto8" is at http://topo.math.u-psud.fr/~lcs/Kto8/. See also the CTAN archive.

Crash course on Russian ASCII-Cyrillic

The 33 letters of the modern Russian alphabet, in alphabetic order, are typed:

```
a b v g d e 'o 'z z i j k l m n o p
r s t u f x 't 'c w 'w q y h 'e 'u 'a
```

The corresponding Cyrillic glyphs are:

```
а б в г д е ё ж з и й к л м н о п
р с т у ф х ц ч ш щ ъ ы ь э ю я
```

Similarly for capital letters:

```
A B V G D E 'O 'Z Z I J K L M N O P
R S T U F X 'T 'C W 'W Q Y H 'E 'U 'A
```

correspond to:

```
А Б В Г Д Е Ё Ж З И Й К Л М Н О П
Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я
```

It is worth comparing this with the phonetic recitation of the alphabet (in an informal ASCII transcription):

```
ah beh veh geh deh yeh yo zheh zeh
ee (ee kratkoe) kah el em en oh peh
err ess teh oo eff kha tseh cheh
shah shchah (tv'ordyj znak) yerry
```

```
(m'agkij znak) (e oborotnoe) yoo ya
```

where parentheses surround descriptive names for letters that are more-or-less unpronouncable in isolation.

When there is a competing ergonomically "optimal" choice for typing a Russian character, the alternative may be admissible in ASCII-Cyrillic. Thus:

```
'g='z
's=w
c='t
'k=x
```

Incidentally, the strongest justification for typing "c" for a letter consistently pronounced "ts" is the traditional Russian recitation of the Latin (ASCII) alphabet:

```
ah beh tseh deh ...
```

For the Ukrainian Cyrillic "hard g" (not in the modern Russian alphabet), Russian ASCII-Cyrillic requires typing:

```
'{gup}
```

(and '{GUP} for the uppercase form). Similarly for other Cyrillic letters. The braces proclaim a Cyrillic letter and the notation is valid for every Cyrillic language.

For the Russian number character, which resembles in shape the pair "No", ASCII-Cyrillic uses the notation

```
'[No]
```

Similarly for the numerous other non-letters. Exceptionally, for this widely used symbol, the short form 'N is allowed. The square brackets proclaim a non-letter. One oddity to note is that for text double right quotes one types `'["]` (4 characters) and not `'['']` (5 characters) while for text double left quotes one types `'[``]` (5 characters) .

The two long notation schemes `'{...}` and `'[...]` afford a systematic way to represent all characters typed on any Cyrillic computer keyboard; and they leave room for future evolution.

The ASCII-Cyrillic expression for an octet >127 not encoded to any normalized character, is

```
!__xy
```

Here `__` is two ASCII underline characters, and xy is the two-digit lowercase hexadecimal representation of the octet. Imagine, for example, that, in the 8-bit Cyrillic text encoding, the octet number hex 8b (= decimal 139) is for non-text graphic purposes or else is undefined. In either case, it is rendered in conversion to ASCII-Cyrillic as

```
!__8b
```

Conversion from this back to the 8-bit form will work. However, although the 5 octet string "!__8b" is ASCII text, this text is not independent of 8-bit encoding. Thus, it is important to eliminate such "unencoded" or "meaningless" octets. A Cyrillic text file containing them is in some sense "illegal".

Tha ASCII letters are the English unaccented letters. The ASCII non-letter characters, namely:

```
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
[ \ ] ^ _ `
{ | } ~
```

are all common to Russian and English computer keyboards and 8-bit encodings. It is worth remembering these non-letters, since you can then identify ASCII text at sight. All of these, except on occasion `' ! \ ,` can be freely used in ASCII-Cyrillic typing of Russian prose; they are not altered under conversion to an 8-bit encoding.

ASCII-Cyrillic is not well designed for typing English sentences, but, since occasional English words or letters are used in Russian, ASCII-Cyrillic allows one to type, for example, `!U` for an isolated `U` and:

```
!Coca-!Cola          for        Coca-Cola
```

The special relationship with TeX

The converter "email-ru.tex" is programmed as a TeX macro package because TeX is perhaps the most widely and freely available utility that can do the job.

The relation with TeX runs deeper. TeX is a powerful stable and portable formatting system, and perhaps the most widely used system for scientific and technical documents. For a continental European language with an accented Latin alphabet (French for example), a TeX typescript is often created as an 8-bit text file that (just as for Russian) depends on 8-bit encoding. However TeX itself has always offered an alternative more prolix ASCII form for such accented letters. For example, `\'e` represents `e` with an acute accent. This ASCII form has always served to provide portable ASCII typescripts that are readable and editable. ASCII-Cyrillic seems to be the first ASCII scheme to offer something similar for all Russian TeX typescripts.

To let users type TeX commands with reasonable comfort in ASCII-Cyrillic, the latter preserves TeX control sequences like `\begin`. The familiar command

```
\begin{document}
```

is thus expressed as:

```
\begin{!document}
```

Russians use mostly ASCII letters in math mode. According to the usage of `!` already explained, the ASCII-Cyrillic $!e=!mc^2$ converts to Einstein's formula $e=mc^2$. The extra exclamation marks are an annoyance. However, for the same TeX output, you could type this formula without the extra exclamation marks --- provided you first define special `\mathcode` values for the octets of `\cyre`, `\crym`, `\cyrc`, etc. Consult the TeXbook about `\mathcode`.

The escape characters: The special roles played by the three characters `' ! \` impose a few strange rules in ASCII-Cyrillic typing. Notably, the ASCII prime `'` must sometimes be typed as `''` (two primes). Experimental use of "email-ru.tex" will allow the user to find his way as quickly as would detailed documentation. (Please report any needlessly complex or absurd behavior!)

Ukrainian ASCII-Cyrillic

This is similar to but distinct from the Russian mode and is not compatible with it.

The 33+1 letters of the modern Ukrainian alphabet, listed in alphabetic order are:

```
а б в г ґ д е є ж з и і ї й к л м
н о п р с т у ф х ц ч ш щ ю я ь  '
```

and the preferred Ukrainian ASCII-Cyrillic form is:

```
a  b  v  g  'g  d  e  'e  'z  z  y  i  'i  j  k  l  m
n  o  p  r  s  t  u  f  x  't  'c  w  'w  'u  'a  q  '*
```

The 34th character is a Cyrillic apostrophe, a "modifier letter" that has various roles, among them those of the hard sign of Russian. The representation valid for all Cyrillic languages is '{apos}.

The phonetic recitation of this alphabet (using an informal ASCII transcription) is:

```
ah beh veh heh geh deh eh yeh
zheh zeh y(?) ee yee yot kah el
em en oh peh err ess teh oo eff kha tseh
cheh shah shchah yoo ya (m'akyj znak)
(apostrof)
```

The alternative short forms in Ukrainian ASCII-Cyrillic: are

```
h=g  's=w  c='t  'k=x
```

The following four letters do not occur in Russian:

```
ґ є і ї

<=>  '{gup} '{ie} '{ii} '{yi} (all Cyrillic languages)
<=>  'g  'e  i  'i   (short forms for Ukrainian)
<=>  (no Russian short forms)
```

Reciprocally, the following four Russian letters do not occur in Ukrainian:

```
ъ ы э ё

<=> '{hrdsn} '{ery} '{erev} '{yo}  (all Cyrillic)
<=> (no Ukrainian short forms)
```

```
<=> q  y  'e  'o     (short forms for Russian)
```

The following two letters are common to Ukrainian and Russian, but the ASCII-Cyrillic short forms are different.

```
    И  Ь
```

```
<=>  '{i}  '{sftsn}    (all Cyrillic)
<=>  y  q     (short forms for  Ukrainian)
<=>  i  h     (short forms for  Russian)
```

In Ukrainian ASCII-Cyrillic, the use of q as a short form for '{sftsn} is supported by the fact that the shape q rotated by 180 degrees is similar to that of '{sftsn} . But there is another reason for this choice. It permits one to use h as an alternative Ukrainian short form for '{g} --- which is natural since in many cases '{g} is pronounced like the harsh German h in "Horst".

Similarly for capital letters. In particular:

```
А Б В Г Ґ Д Е Є Ж З И І Ї Й К Л М
Н О П Р С Т У Ф Х Ц Ч Ш Щ Ю Я Ь '
```

have the Ukrainian ASCII-Cyrillic representation:

```
A  B  V  G  'G  D  E  'E  'Z  Z  Y  I  'I  J  K  L  M
N  O  P  R  S  T  U  F  X  'T  'C  W  'W  'U  'A  Q  '*
```

Long forms valid for all Cyrillic languages are:

```
'{A} '{B} '{V} '{G} '{GUP} '{D} '{E} '{IE} '{ZH} '{Z}
'{R} '{I} '{II} '{YI} '{J} '{K} '{L} '{M} '{N} '{O}
'{P} '{S} '{T} '{U} '{F} '{X} '{TS} '{CH}
'{SH} '{SHCH} '{YU} '{YA} '{SFTSN} '{APOS}
```

Note that the Ukrainian apostrophe '{APOS} is a **letter** and, unlike '{SFTSN}, it normally coincides with the lowercase version: normally '{APOS}='{apos}. In case of a distinction, '* will be '{apos}. Further, '{apos} normally has shape identical to the text right single quotation mark denoted in ASCII-Cyrillic by '[].

There is an official lossy ASCII transliteration for Ukrainian using the ligature concept, and it is supported by "email-ru.tex". See the Ukrainian national norm of 1996 summarized at:

```
http://www.rada.kiev.ua/translit.htm
```

Beware that the official Ukrainian transliterations of the six letters:

```
'{g}   '{ie}   '{yi}   '{ishrt}   '{yu}   '{ya}
```

are context dependent. This is a good reason for calling upon "email-ru.tex" to do the official transliteration.

The other aspects of ASCII-Cyrillic are the same for Ukrainian and Russian.

Vital statistics for ASCII-Cyrillic

ASCII-Cyrillic home page: (established December 2000)

```
topo.math.u-psud.fr/~lcs/ASCII-Cyrillic/ascii-cy.htm
```

ASCII-Cyrillic software directory:

```
http://topo.math.u-psud.fr/~lcs/ASCII-Cyrillic/
```

Long term archiving: See the CTAN TeX Archive and its mirrors.

Copyright conditions: Gnu Public Licence.

Documentation: -currently included as ASCII text inside the converter "email-ru.tex".

Debts: The author owes many thanks, in particular:
-- to Stanislas Klimenko for an invitation to IHEP Protvino, Russia, in Fall 1977; ASCII-Cyrillic was conceived there;
-- to Irina Maxova'a for suggesting in November 1997 that the ASCII `w` represent the Cyrillic letter "sh" (`\cyrsh` in TeX);
-- to Gal'a Gor'a'cevskix for answering innumerable questions about Russian;
-- to the members of the Cyrillic TeX discussion list (CyrTeX-en@vsu.ru), moderated by Vladimir Volvovi'c, both for clarifying problems and for furnishing vital data. The

list archives are available at:

`https://info.vsu.ru/Lists/CyrTeX-en/List.html`

-- to Maksym Pol'akov (mpoliak@pcomp.nauu.kiev.ua) whose extensive advice was essential in establishing Ukrainian mode.

Date of most recent modification: July, 2001.

The author: (who welcomes comments)

Laurent Siebenmann
CNRS, Université de Paris-Sud
Orsay, France

lcs@math.u-psud.fr
lcs@math.polytechnique.fr
laurent@math.toronto.edu

♢ ♢ ♢

A Tour around the $\mathcal{NTS}$ implementation

Karel Skoupý

ABSTRACT. $\mathcal{NTS}$ is a modular object-oriented reimplementation of TEX written in Java. This document is a summary of a presentation which shows the path along which the characters and constructions present in the input file pass through the machinery of the program and get typeset. Along the way the key classes and concepts of $\mathcal{NTS}$ are visited, the differences with original TeX are explained and the good points where to dig into the system are proposed.

KEYWORDS: $\mathcal{NTS}$, Java, extension

$\mathcal{NTS}$ is a modular object-oriented reimplementation of TEX. It is written in Java and is meant to be extended with new functionality and improvements. In spite of the expectations of many it is not simpler than original TEX and it probably could not be if it has to do exactly the same job. But whereas TEX is a very monolithic system, the complexity of $\mathcal{NTS}$ is divided into many independent modules and is accommodated in lots of useful abstractions. As a consequence one need not know all the details about the whole system in order to extend or change just a specific part. The dependencies between the modules are expressed by clear interfaces and the interfaces is all one needs to know about untouched parts. So extending and changing $\mathcal{NTS}$ should be quite easy, shouldn't it?

The problem is that detailed documentation is still missing and that there are hundreds of classes, so it is hard to know where to start. Although the classes are many, fortunately there are a limited number of main concepts which are really important.

The processing in $\mathcal{NTS}$ is naturally quite similar to TEX. The input is scanned and converted to *Tokens*. Each *Token* has a certain meaning: a *Command*. Some *Commands* are non-typographic, these usually deal with macro expansion or registers. Typographic *Commands* build some typographic material consisting of *Nodes* using *Builders*. Lists of *Nodes* are packed by a *Packer* and finally typeset by an instance of *Typesetter*.

Of course there are a few more basic concepts than those emphasized in the previous paragraph but not that many. They are always represented by an abstract class or an interface. The other classes in $\mathcal{NTS}$ are either various implementations of those interfaces or they are auxiliary and not so interesting.

Recently we have been trying to improve the design of $\mathcal{NTS}$ so that extensions and configuration are even easier. We will also look into ways how to increase performance and interoperability with the TEX directory structure.

♢ ♢ ♢

Visual TEX: texlite

Igor Strokov

abstract. A prototype of a visual TEX is implemented by means of minor modifications of canonical TEX. The changes include the ability to start compilation from an arbitrary page, fast paragraph reformatting, and retaining the origin of visual elements. The new features provide direct editing of the document preview and correct markup of the source text.

keywords: visual, TEX

The need for visual TEX

A good feature which TEX traditionally has lacked is visual editing, or the ability to typeset a document in its final (print preview) form. Though one can manage without visual editing, there are certain cases when it really helps, mainly when tuning the appearance of a document, especially for novice and occasional users.

There are two ways to deal with this problem. The first one, implemented in lyx and SciWord, represents the source text in a form which logically resembles the resulting output. Although this way proved to be a good compromise, the logical preview is often quite far from the printed result. Besides, these tools use latex with special macros, so arbitrary TEX documents are beyond their scope.

The other way is synchronously running a source text editor, TEX the compiler, and a viewer, so that changes made in the source editor are compiled and displayed in the viewer without explicit invocation of these tools in a command string or a menu. In textures, in addition, the cycle is closed by means of two corresponding pointers in the source editor and the viewer. The approach of Jonathan Fine, presented in these proceedings [2], can be related to this way. Its main problem lies in the need to compile a whole document to receive the visual response to an editing action. On slow computers, large documents, or complicated macros this compilation could cause at least a noticeable delay.

How to TEX part of a document

So, if even a single letter is changed, TEX needs to process the whole document from scratch. However, TEX already solved a similar problem by loading precompiled macros (so called 'formats'). This way TEX saves the time required to compile standard

macros which can be considered a part of a document. With minor changes one can extend this method to arbitrary stages of the document compilation. One only need to choose particular stages and probably improve the storage method.

The natural decision is to make a core dump each time a typeset page has been completed and removed from TEX's memory. At this moment the memory is relatively empty and contains essentially values which do not change from page to page, which allows the dump to be more compact. A page dump is always stored as a record of differences from some reference dump. Every 8-th dump refers to dump zero (the 'format' itself) while the others refer to the preceding 8-th one (see [3] for details).

In comparison to a common format the core dump has to store additional data: the input and semantic list stages, open file pointers, and, above all, the source line number reached when the page was completed. Thus, if the source is changed at some line, one can derive the last page number affected by the change. Then the corresponding dump is loaded and the compilation starts from the given page.

In texlite, the visual TEX prototype, the compilation is allowed to run up to the end of the document. Moreover, if any output file to be read again is changed, then the compilation starts again, this time from the very beginning (this happens, for example, when a LATEX section title is edited after the table of contents). However, this already does not matter, as the page of interest is obtained quite quickly (in constant time, regardless of the page number and the document size). Although in some mentioned cases the page is updated a second time, in practice this event does not disturb a user because of buffered output and preservation of the current input position.

Besides, if another source change will occur before the natural end of the compilation, it will be interrupted to let a more actual compilation run. As a result, all recent editing actions are reflected in time, while the remote consequences (if any) appear after a pause.

How to reformat a paragraph

Selective compilation is a necessary but not sufficient feature for visual editing. It just reduces the response time from the source to the preview (making it constant instead of linear). Let us consider the inverse problem — how to bring preview changes into the source text.

In general, one needs to keep track of the relation of source characters to visual elements. In many cases the reciprocal relation (say, a character — a glyph) may be established and used to synchronize current positions in the source and the preview. Fortunately, no vast interference in TEX is needed to make it remember the origin of its output: it is enough to extend the format of memory nodes in a way similar to that used for breaking the 64K barrier.

So, a user may mark the current position in the preview and type something there. Texlite applies the corresponding changes in the source text and initiates the compilation starting from the current page. In most computers (starting from a 200MHz Pentium) the delay before the visual result is virtually unnoticeable. However, one cannot ignore slow computers and various decelerating factors such as parallel pro-

cesses, complex page formatting, complicated macros, etc.

Therefore, before starting the true compilation, texlite tries to reformat the current paragraph using the native TEX algorithm for this purpose. Canonical TEX, however, does not keep parameters it used to set up boxes and paragraphs. That is, one cannot correctly rebuild a box or a paragraph only from its contents. Texlite resolves this problem by storing the necessary data in special *whatsit* nodes. This does not take too much extra space, as many parameters (penalties, glues, *parshape*, etc.) remain the same throughout a document and thus can be omitted.

Let us see what happens when a user edits the preview. First of all, texlite decides (with the aid of whatsit nodes) in which paragraph, if any, the current position falls. If no paragraph is recognized (for example, it may be within a `\halign`), then only the enclosing box is rebuilt and the selective compilation starting from the current page is initiated. Otherwise texlite locates the current paragraph and unwraps it back into a *hlist* by inserting lost glues and repairing hyphenations. The unwrapped list is subjected to the changes following from the user's input (insertion of a character-and-glue node list or deletion of several nodes from the current position) and the *linebreak* routine is called to rebuild the paragraph. Then the preview (along with the current position) is updated.

After this 'emergency repair' texlite enters the source text, performs parallel changes there and starts the selective compilation, which runs from the current page to the end of the document or until the user presses another key. If the compiler manages to build the current page before the next key stroke (usually it does) and the new page happens to be different from the repaired one (usually it does not) then the preview is accurately updated.

Implementation

Texlite, currently implemented under Win32, provides a visual shell for the TEX core tangled from TEX: The Program [1] and modified in the way described above. Let us enumerate the main changes:

1. It can make detailed core dumps and read them back at specified points.
2. For every paragraph it stores all the data required to unwrap the paragraph back and break it into lines again.
3. It relates nodes in memory to their origin in the source text.

Except for these additions the compiler remains the canonical TEX, able to process an arbitrary TEX document.

The shell consists of two windows. The preview window represents the current page in its typeset form, which is common to all DVI viewers. The main difference is an editing cursor (a flashing caret). The user may mark, type, or delete pieces of formatted text as in any WYSIWYG text processor, without delays or other inconveniences. However, the requirement to keep a correct document structure still imposes some limitations. For example, one cannot select a part of a heading and several words in a following paragraph and delete them at once. Texlite prevents such inconsistent

changes by tracking grouping levels.

Texlite preview supports graphics in several bitmap formats and in PostScript (installed GhostScript required). Standard LATEX coloring schemes and emTEX extensions (popular in the DOS/Windows community) are supported too.

In addition, texlite provides an extension for internal and outer links in a document. For example, a specific macro makes a common table of contents into an interactive directory which opens the corresponding page when a section reference is clicked. An outer reference brings a user to a different document located elsewhere on Internet. This feature facilitates browsing of TEX documents and makes it similar to web surfing.

Though a user may choose to work with a document in the preview window only, there remains a window for the source text. All the changes made in the preview are automatically reflected here. However, if a user edits the source text, he should invoke the synchronization explicitly, because manual correction of the source text allows the existence of inconsistent clauses.

The source text view benefits from visual editing too, as it uses information from the compiler to mark up the text according to the TEX syntax. Moreover, the markup (presented in different colors) is always correct, even if symbol categories are changed during the compilation.

A small improvement concerns error corrections. The shell sets the cursor to the position in the source text where an error occurs and displays its description. However, if a compiler is used just to mark up the text, errors are only marked by a color.

REFERENCES

[1] Knuth, D. E. Computers & Typesetting. Volume B, TEX: The Program. *Reading, Massachusetts: Addison-Wesley, 1986.*

[2] Fine, J. Instant Preview and the TEX daemon. *These proceedings.*

[3] Strokov, I. I. A WYSIWYG TEX implementation. *TUGBoat*, December, 1999.

Conversion of TEX fonts into Type 1 format

Péter Szabó*

ABSTRACT. This paper analyses the problem of converting TEX fonts to Type 1 fonts, describes TEXTRACE, a new free conversion program, and compares it to other possible methods and existing utilities. TEXTRACE works by rendering the font in high resolution and then tracing (vectorizing) it.

KEYWORDS: PDF, font conversion, Type1, METAFONT, vector, outline, raster, bitmap, pdfTEX

THERE ARE two major ways to describe the shape of the glyphs[1] of a font: using *bitmap images* or *vector outlines.* A bitmap image is a rectangular array of dots; each dot can be painted white or black independently from the others. At a high resolution of around 600 DPIthe human eye cannot distinguish individual pixels and sees the glyph as if it was continuous. On the other hand, vector outlines specify the black regions by the curves (lines, arcs, ellipses and others) which delimit them. The most important advantage of this representation is that glyph outlines can be transformed (rotated, scaled etc.) without quality loss.

Most fonts used in today's desktop publishing systems are distributed in some kind of vector outline format. (Such formats are: Type 1, TrueType, OpenType, Speedo, METAFONT source.) The process of converting an outline font to its bitmap equivalent is called *rendering* or *rasterization.* The rendering must surely happen before actually printing the document onto paper or displaying it on screen, because today's printers and screens display everything using a rectangular layout of dots. So the difference between document preparation systems lies not in the presence of the rendering, but the moment when it occurs.

When printing a document onto paper, the exact time instant of the rendering is not important ~ the author wants the final output be beautiful, and all other inner parts of the printing process are of minor importance. However, the case is different when the document is about to be distributed in electronic form. Since readers might use different software for viewing and/or printing, the author should ensure that the document looks and prints nice in most environments, i.e. the document should be

*Many thanks to Ferenc Wettl and Gyöngyi Bujdosó

[1] *glyph:* a specific, visible representation of a character

created to be portable. This involves selecting a font format that works well in most reader software, and using this font format exclusively in electronically published versions of the document.

In our specific case, the file format used for document publishing is PDF, and one of the best font formats for embedding is *Type 1,* a traditional vector outline format. (These will be presented later in more detail.) Today's TEX-to-PDF generating software support Type 1 fonts, but unfortunately most TEX fonts are currently not available in Type 1 format. So they have to be converted.

This paper focuses on a quite difficult font conversion problem. The most common source format, METAFONT is rather complicated (in fact, it is an almost full-featured programming language), and the destination format, Type 1 has too many restrictions. Probably this is the reason why there have been no good and robust programs to do the conversion. TEXtrace, described later in this document, overcomes the first problem by not even trying to understand the METAFONT files, but letting the original METAFONT program render the glyphs, and then calling the AutoTrace program which traces the bitmaps (approximates them with vector outlines). Fortunately AutoTrace's output needs only minor modifications to comply the restrictions of the Type 1 format. The main disadvantage of this "blind" conversion approach is a minor quality loss caused by both the rendering and the tracing process.

The availability of the fonts in a vector outline format is of primary importance when publishing the document as PDF, because Acrobat Reader, the most widely used PDF viewer displays bitmap fonts slowly and ugly on screen.

Motivation

Why PDF?

The PDF file format is very popular nowadays among authors to distribute their work electronically. PDF has become popular because it

- represents letters, spaces, line breaks and images etc. accurately: documents retain their structure and original visual appearance between PDF viewers (as opposed to markup languages, such as SGML and HTML).
- follows the industry standard PostScript page drawing model, PostScript files can be converted to PDF without quality loss (and vice versa).
- is portable and device-independent. Printing PDF files and rendering them on screen is often easier and more straightforward than doing the same for PostScript files. It is possible to create a portable PDF document, but one can never be sure about PostScript portability.
- has widespread freely available viewers (e.g. Acrobat Reader: either stand-alone or in the web browser, Ghostscript, xpdf, PDF e-Book readers) for most platforms. Most systems have some sort of PDF viewer installed.
- has hyperlink, form submit and other web-related capability. (This article ignores these facilities and deals only with static PDF files.)

FIGURE 1: THE DIFFERENCE BETWEEN VECTOR (NICE) AND BITMAP FONTS IN ACROBAT READER

- allows relatively small file sizes, which is comparable to DVI files.
- is well suited for both short and long documents, including scientific articles, reports, books and software documentation.

Of course, it is possible to convert TEX output into PDF format either by calling pdfTEX [6], which directly creates PDF files instead of DVI files, or using a DVI-to-PDF or PostScript-to-PDF converter. Preparing a PDF version of a TEX document is somewhat harder than just printing the PostScript version, and requires a bit more time. To enjoy the benefits listed above, the author must use the proper version of the utilities, she must adjust the settings, and she must make sure that the proper versions of the fonts get embedded.

Font problems with PDF and TEX

The main problem with PDF files originating from TEX source is that characters of most TEX fonts show up slowly and look ugly in Acrobat Reader. This is because such fonts are generated by METAFONT, and are embedded into the PDF file as high resolution bitmap images; and Acrobat Reader displays bitmap images slowly and poorly on screen. This has been one of the famous Ugly Bits related to TeX for years. The problem doesn't occur when printing the document, because the resolution of the printer is most often high enough ($\geq$ 300 DPI) to eliminate rendering problems.

The solution to this problem is to embed the offending fonts in such a file format which Acrobat Reader can display quickly and beautifully. A good candidate for this is the classical, standard *Type 1* format, appeared in the late '80s. Fortunately pdfTEX and newer versions of `ps2pdf` (Ghostscript $\geq$ 6.50) fully support it, so the only problem remaining is that how to convert a font in METAFONT format to the Type 1 format. Many techniques have been proposed so far, and none of them is fully satisfactory.

The problem itself can be simply visualized by running `pdftex` on simple `.tex` file (see also Figure 1):

```
Sample text: nice\par \font\f=ecrm1000\f Sample text: ugly\end
```

Both the console output of `pdftex` and the `.log` file should contain the entries `ecrm1000.600pk>` and `<cmr10.pfb>` (the number 600 may vary across installations). These entries are the names of the two font files `pdftex` has embedded into the PDF file. The `pk` extension means that the glyphs of the font have been embedded as bitmap images (the resolution of 600 DPI is also indicated ~ this is of course printer dependent), and the `.pfb` extension means that the glyphs have been included as vector outlines in the Type 1 format.

Differences between METAFONT and Type 1

We'll focus only on how these formats describe the shapes of individual glyphs. The difference between how font-wide metrics, glyph metrics, kerning and ligatures are treated isn't much important, because this information can be converted easily and precisely -- and this information will be acquired from the original TEX fonts (the `.tfm` files) anyway, so we don't have to convert them at all.

Taco Hoekwater gives a detailed, richly illustrated comparison of the two font formats, including many figures and a case study of converting some math and symbol fonts (e.g. `logo8`, `wasy5`, `stmary7`, `rsfs10`) in [3]. The curious but beginner reader is referred to that article.

The fundamental difference between the METAFONT and Type 1 is that METAFONT is a programming language, and Type 1 is just a compact file format describing glyph data[2]. The compiler/interpreter of the METAFONT language is the `mf` program, which reads the font program source from `.mf` files (written by humans). The METAFONT programming language contains -- among others -- line and curve drawing instructions. `mf` follows the instructions in the font program, and draws the glyphs into a bitmap in memory, and saves the bitmap into a file when ready. Thus the printable/embeddable version of a METAFONT font is available only in bitmap format -- the rasterization cannot be delayed.

METAFONT programs describe lines and curves indirectly: they impose mathematical equations the shapes must fulfill, and `mf` determines the exact shapes by solving these equations. For example, the font designer is allowed to define a point as an intersection of two, previously defined lines, and to leave the calculation of the exact coordinates to `mf`.

On the other hand, Type 1 fonts define points and shapes calculated previously, in the time of font creation. This suggests that a Type 1 font is the final output of some font generation process, such as a compiler similar to METAFONT (see [4]), or -- more commonly -- the export facility of a graphical, WYSIWYG font editor[3].The process itself may be arbitrary as long as its output conforms to the syntax described in the Type 1 documentation [1]. The font contains the glyphs as vector outlines, in the form of curves consisting of straight lines and Bézier-curves. The rendering (rasterization) into the printer-specific resolution is deferred as much as possible; thus Type 1 fonts are scalable (resolution-independent). Type 1 doesn't define a specific rendering algorithm, so minor pixel differences might occur when printing the same document containing Type 1 fonts on different printers. The rendering algorithms are relatively simple (much simpler than an `mf` implementation), and they are fast enough for real-time operation: each instance of the glyph is rendered independently to the others by the printer[4]. The Type 1 file format relates closely to the PostScript programming language (the font file is a valid PostScript code in fact), but it is possible to render Type 1 fonts without knowing how to interpret PostScript.

Type 1 fonts can be easely converted to METAFONT format (see [4]). This conversion

[2] a similar difference exists between TEX and HTML; one cannot write a loop that prints n stars in pure HTML, but it is possible in TEX

[3] such as the free and sophisticated PfaEdit program [8]

[4] this is not exactly true for modern printers that have a glyph cache

is merely syntactical: similar to converting plain text into a program that just prints the text into its standard output and quits: we just convert a glyph outline to METAFONT drawing instructions that draw the outline. However, this direction of conversion is quite useless, because TEX can already deal with Type 1 fonts well (even for PDF files), plus the inner structure of the glyphs wouldn't be revealed, so we'd lose the benefit of the geometrical (programmed) font generation philosophy of METAFONT.

The other way, converting `.mf` to Type 1 is expected to be harder, because the METAFONT language is much more complicated than the Type 1 syntax. So we cannot convert the METAFONT programs exactly without "understanding" them. Because current `.mf` files tend to be quite complicated, full of dirty tricks, and they are utilizing most features of the language, the only reliable way to understand these files is using a full METAFONT interpreter. There are two such interpreters available: `mf` itself and `mpost` (METAPOST). `mf` doesn't fit our needs, because it generates bitmap images and not vector outlines. On the other hand, `mpost` generates EPS (Encapsulated PostScript) output, containing vector outlines which are very similar to the Type 1 outlines. But, due to the restrictions of the Type 1 format, these outlines must be post-processed. This cleanup process is so complicated that currently *no* program exists that can do it reliably. Alternative ways of using of METAFONT or METAPOST for conversion will be discussed two sections later.

Of course ⇝ as in the case of TEXTRACE ⇜ one may go ahead without trying to understand the METAFONT language, and looking at only `mf`'s bitmap output. These bitmaps must somehow be converted (traced) into vector outlines that approximate the bitmap. Fortunately there exist automatic utilities for this.

A tour from CM to EC

TEX's default and most common font family is the Computer Modern (CM) family, designed by the author of TEX and METAFONT, Donald Knuth himself, between 1979 and 1985[5]. The fonts were available only in METAFONT format till 1992, when BlueSky converted them to Type 1. Now the Type 1 `.pfb` files are available from CTAN and they are part of most standard TEX distributions (such as teTEX and MikTEX).

pdfTEX is configured to use these `.pfb` files by default (`pdftex.cfg` includes `bsr.map` which contains all the available font names), so there should be no problem when creating PDF from those TEX documents which use the CM fonts.

The EC abbreviation stands for the European Computer Modern family, designed by Jörg Knappen between 1995 and 1997. *European* means that these fonts suit typesetting in some non-English European languages better than the CM fonts. Apart from this, the EC fonts are basically the same as the CM fonts with a couple of minor additions and adjustments. It is almost impossible to distinguish a font in the EC family from its CM counterpart[6]. Tha main difference between these two families is technical: the EC family contains the accented characters directly in the font, while the CM family contains the letters and the accents separately and TEX macros are used to position the accents. The EC fonts are thus larger, but they have an important

[5] for those who are still wondering: this paper uses the Computer Modern fonts almost exclusively.

[6] one of the visible differences are the accents; for example the CM acute accent is taller, and the CM Hungarian umlaut accent is more angled than its EC counterpart

benefit: text entered using those fonts can be automatically hyphenated by TEX (in the case of CM fonts the macros prevent automatic hyphenation).

Unfortunately the EC fonts are not available in Type 1 format yet, and most European TEX users are suffering from this problem. The primary reason why TEXTRACE was written is to do the conversion. However, several methods existed before TEXTRACE to overcome this Ugly Bit. Some of them are:

- The AE fonts with the LATEX package `ae` simulate the EC fonts from the available CM fonts with virtual fonts[7]. Both requirements are fulfilled: automatic hyphenation works because the virtual font gives TEX all the accented glyphs, and PDF files look nice because the virtual font points to the CM fonts, already available in Type 1 format from BlueSky.

  The most serious problem of the AE fonts is that only the most commonly used EC fonts are simulated, and they are available only in some of the EC design sizes (5–8 of 14). Of course some glyphs are missing because they were missing from the CM fonts (examples: eth ð, gulliemots « »). Another problem is that the metrics are a bit different from the original font, and this affects line and page breaks of the document. (This could be easily solved by overwriting the `.tfm` files from the good EC fonts.)

  Correcting the accents and creating all design sizes would have been possible with a couple of weeks of work, but the missing glyphs imposed an unsolvable limitation. And the method worked only for the EC fonts, it couldn't cope with all TEX fonts.
- The ZE fonts are similar to the AE fonts, but they collect the missing glyphs from common Adobe fonts (such as /Times-Roman). Apart from this, they have the same fundamental problems as the AE fonts.
- Automatic hyphenation works well when typesetting an English (and other latin non-accented) document. The already available CM fonts can be used for this. In LATEX, the only thing the author must ensure is that the document preamble *doesn't* contain `\usepackage{t1enc}`.
- There are language-specific conversion utilities which modify `.tex` source files and insert discretionary hyphens (`\-`) into the appropriate places. So the hyphenation problem is solved, the automatic hyphenation provided by TEX isn't used at all. These utilities are not part of TEX itself, and have different interface for different languages. They contradict the generic, consistent and accurate way as Babel works, and make typesetting with TEX more difficult and error-prone. The solution is font-independent, so it works with the CM fonts.

Thus, in the case of the Computer Modern family, one can use the tricks above to manage without the Type 1 versions of the EC fonts. But there are no such tricks for most other TEX fonts: they need conversion, and for the conversion, one needs a proper converter.

[7] a TEX virtual font builds its glyphs from glyphs of other fonts. Virtual fonts can be used for re-encoding and/or collecting glyphs from several fonts.

TEXTRACE

TEXTRACE is a collection of scripts for UNIX that convert any TEX font into a Type 1 `.pfb` outline font immediately suitable for use with DVIPS, pdfTEX, Acrobat Reader (and many other programs). The documents using these fonts cannot be visually distinguished from the documents using the original TEX fonts, moreover the PDF documents display quickly and nicely in Acrobat Reader.

The main goal for writing TEXTRACE was to convert the EC fonts to Type 1 format with it. This goal has been fulfilled, 372 of the 392 font files (28 fonts · 14 design sizes) are available as `.pfb` files with an average file size of 86 kb. The missing 20 files are the Typewriter fonts at 5pt and the experimental Fibonacci and Funny fonts at various sizes. They are missing because METAFONT failed to generate them. The conversion took 40 hours with a 333 MHz Celeron processor and 128 Mb memory.

The 392 TC (Text Companion) fonts and some others have been also converted. (With some failures, because of METAFONT overflow or font program error).

TEXTRACE is in beta status, the author is waiting bug reports from users.

How it works

TEXTRACE renders all the 256 glyphs of the TEX font in a very high resolution ($\geq$ 7000 DPI), then it converts the resulting bitmaps to individual vector outlines, and finally it assembles these outlines to form a syntactically correct Type 1 font. The final result is a `.pfb` font file, which can be read by both DVIPS and pdfTEX to be included into documents.

The principle sounds simple, but the implementation is quite complicated because of the usual quirks of programs used, incompatible software and loose (or disobeyed) file format specifications.

TEXTRACE operates fully automatically. It doesn't require human intervention, it doesn't ask questions. The output of TEXTRACE is useful immediately, needs no further conversion, re-editing or adjustment in font editors. TEXTRACE can operate in batch mode; this means converting all the fonts listed in a `.map` file.

The operations of TEXTRACE for a single font:

1. A `.tex` file is generated, each page with a different glyph from the font.

   Not only the shape of the glyph is important, but the absolute coordinates of the character origin must also be specified: this guarantees that glyphs will line up to basepoint and follow each other the same distances in the original and the converted font. So the TEX primitive `\shipout` is used to output pages. This way page numbers and the various glues are completely avoided, so the glyph is typeset to a well-defined, exact place on the page.

   The font is requested at a relatively large size (120.45 pt for 600 DPI and fonts with design size of 10 pt), so that 1 *em* will be 1000 pixels long when rendered.

   TEX is instructed to output glyph metrics into a separate `.log` file.

2. `tex` is invoked to create the `.dvi` file from the `.tex` file. This is a quite simple job for TEX: no line or page breaks etc. occur. As a side effect, `tex` runs `mktextfm` (or whatever font generation mechanism is used in the TEX distribution) if the `.tfm` file was missing.
3. `dvips` is run, which converts the `.dvi` file to `.ps` and automatically embeds the font into the `.ps` file.

   DVIPS reads the font from the appropriate `.pk` file (e.g. `ecrm1000.7227pk`) most of the time. If the `.pk` doesn't already exist (which is probably true since the chances for a previous use of the font at 120.45 pt or so are negligible), it calls `mktexpk` to create it. `mktexpk` looks around and passes the job to `mf`.

   All this happens automatically, without TEXTRACE having to know about TEX font generation and embedding internals.
4. `gs` (GNU Ghostscript) is called to render (rasterize) the `.ps` file.

   This is almost a syntactical conversion, because METAFONT fonts are already inserted into `.ps` file in rasterized bitmap `.pk` format by DVIPS[8]. Of course the output bitmaps of Ghostscript are not sent to the printer, but they are saved to individual image files (in the PBM format).

   Special care must be taken for glyphs with negative sidebearing (i.e. glyphs extending to the left from their origin). For normal printing, this isn't a problem, because a sub-millimeter negative sidebearing is negligable with a left margin of 2 cm. In the case of TEXTRACE, we must make sure that the glyph won't stick out of the page. This could be accomplished by moving a large (but constant) amount to the right and hoping that the left sidebearing is smaller than that amount. TEXTRACE uses a more precise approach: it measures the glyph bounding box (with a careful trickery using the `bbox` device of Ghostscript), and moves the appropriate amount to the right.
5. The individual bitmaps are converted to vector outlines by the free external program AutoTrace (written by Martin Weber `<martweb@gmx.net>`).

   This converter supports several input and output file formats, but these are mostly equivalent. The EPS format is used for output, because it is very similar to how Type 1 represents glyphs. However, Type 1 imposes some restrictions on the glyphs, so a little modification of the original AutoTrace output routine was required to obey these restrictions:

   - Path directions (clockwise/counter-clockwise) had to be corrected, so that black regions would always be to the right-hand side from the path.
   - Multiple paths had to be converted into a single one.
   - The initial clearing of the image to white color had to be removed.
   - Curves didn't touch each other in the original output, so no modifications were necessary.

   The AutoTrace program has bugs: it errs out with some failed assertion error for 0.12% of the glyphs. The author of AutoTrace is aware of the bugs, but he currently has no fix for them. The bug is worked around empirically, by calling

[8] an exception: when DVIPS has embedded a font in other than `.pk` format.

FIGURE 2: AUTOTRACE COMPARED TO OTHER TRACING SOFTWARE

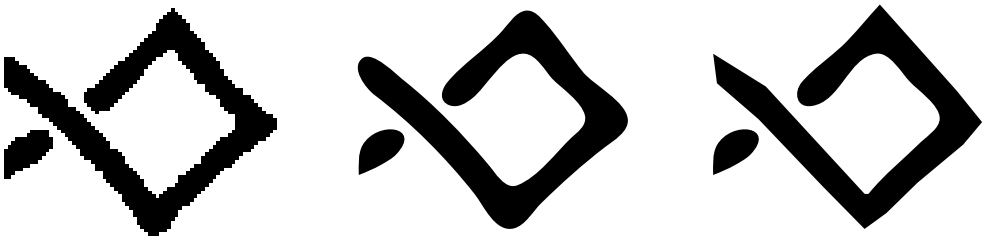


FIGURE 3: AUTOTRACE CANNOT FIND CORNERS AT LOW RESOLUTION

AutoTrace with different parameters until the bug stops to manifest.

AutoTrace – although it is free – isn't currently available from the web because of technical reasons. So the proper version of AutoTrace is included in the TEXTRACE distribution.

AutoTrace was chosen for the tracing because it is free, works automatically, and gives a good result (see Figure 2 for a comparison). Other examples of glyphs traced by AutoTrace are the ornaments throughout this document. AutoTrace does a quite good job at extremely high resolutions (such as the 7227 DPI used by TEXTRACE), but produces poor and useless output for low resolutions (example: when a glyph at 10pt is scanned at 600 DPI).

Figure 3 illustrates that the sharp corners produce the most problems for AutoTrace. Adjusting the command-line options doesn't help, and it might even do harm if different regions of the image require different options. Fortunately such distortions do not occur when using TEXTRACE, because fonts are rendered in a high enough resolution.

6. The vector outlines are cleaned up a bit to comply the Type 1 specification.

   The details of the cleanup have been explained in the previous item. The cleanup after AutoTrace is much easier than it would be after METAPOST (see [3]).

   The outlines are also properly scaled and shifted respect to the origin.

7. The outlines are merged into a Type 1 font file.

   `type1fix.pl` (a utility bundled with TEXTRACE) is run on the font file to fix quirks in common font editors and other font handling software (including pdfTEX and DVIPS).

   The output `.pfb` file will be highly portable and compatible with most font software. In an ideal world (with no Ugly Bits) there should be no font compatibility issues. However, in the real world, such issues are common since there is no standard way to read a Type 1 font apart from interpreting it as a PostScript program. (Of course that'd require a full PostScript interpreter in all font handling programs, which is too complicated and would make execution slow).

So most utilities (including those mentioned above) use a different, nonstandard Type 1 reader (invented by the "common sense" of their programmer). These readers tend to have specific lurking quirks: they are able to load most Type 1 fonts, but they might run into a problem with loading an "offending" fonts, which fully follow the Type 1 specification. Fortunately to output of `type1fix.pl` avoids all common quirks, so fonts generated by TEXTRACE don't suffer from incompatibilities of third-party software.

Availability

TEXTRACE is free software, licensed under the GNU General Public License. It can be downloaded from its home page:

`http://www.inf.bme.hu/~pts/textrace/`

TEXTRACE requires a UNIX system with the teTEX distribution, GNU Ghostscript, Perl and GNU Bash to run. Successful runs have been reported on MacOS X and Win32, but those platforms are not supported. Of course, `.pfb` font files generated by TEXTRACE are platform independent, and they can be used in any TEX environment.

Some fonts converted by TEXTRACE are available for download from the same home page. These include all the EC and TC fonts, some missing AMS fonts and others.

Problems to be solved

- *Huge font files.* The `.pfb` files TEXTRACE generates are around 3.15 times larger than the optimum. This is mostly because of AutoTrace cannot find the optimal Bézier-curves that describe the outline. To solve this, AutoTrace has to be partially rewritten or an other tracer must be used. (The author doesn't know of free tracers comparable to or better than AutoTrace.)
- *Corners get smoothed.* Finding the corners (sharp angles) of the outline is a weak point AutoTrace: sometimes it recognizes corners as small arcs. The author tried adjusting the tracing parameters to solve the problem, but hasn't succeeded.
- *Hinting information is missing.* Type 1 fonts use the so-called *hinting* technology to improve rendering on small resolutions (e.g. on screen, ≤ 100 DPI). This requires global and glyphwise additional information in the font, called *hints*. To achieve high quality, this information should be inserted by experts with great care, the process can be automated only with loss of quality. Fonts generated by TEXTRACE completely lack hinting, so glyphs are expected to be badly readable in low resolutions. Fortunately, in practice, fonts generated by TEXTRACE can be read quite well in Acrobat Reader unless one turns antialiasing off explicitly[9].
- *Conversion is slow.* With a 333 MHz Celeron processor and 128 Mb of memory, conversion of a single font (e.g. `ecit1728`) takes around six minutes. This is tolerable unless one wants to convert hundreds of fonts (such as all the 392 EC fonts). Although this might sound quite slow, currently TEXTRACE is the fastest in its category. Moreover, the process is fully automatic, and the result is immediately

[9] with File→Preferences→General→Smooth...

usable without any further human intervention. Although doubled speed could be achieved with complete rewrite of the code, this wouldn't worth optimizing.

- *Limited portability.* TEXTRACE is requires a UNIX environment, but could be easily ported to other systems (with TEX, DVIPS and Ghostscript) in 1–2 days of work. TEXTRACE consists of programs written in Bash (shell script), Perl, PostScript and ANSI C. The C part cannot be avoided because the engine, AutoTrace is written in C. The PostScript part is OK, because Ghostscript is needed anyway. Shell scripts can be easily rewritten in Perl, and Perl scripts are possible to be rewritten in C, but it would require too much work.
- *Written for experts.* Although the documentation describes the conversion process for beginners, expert level knowledge is needed to enjoy TEXTRACE's benefits. The selection of source fonts, and the proper usage of the destination fonts in documents is left to the user. So is the recovery from errors: TEXTRACE is quite paranoid and stops on most errors, but the error message doesn't always indicate the real cause, and the user is left alone for figuring out and making corrections.
- *Fails for some `.mf` files.* METAFONT uses fixed point arithmetic to make rounding errors identical on all architectures. The bits representing the fractions are chopped from the integer part, so the range of numbers in METAFONT is small. The large rendering resolution asked by TEXTRACE requires large numbers. This results sometimes in overflow errors for some glyphs, and `mf` fails to generate those glyphs. TEXTRACE currently refuses to convert fonts with bad glyphs. The problem is hard to solve in the general case: the implementation of arithmetics in METAFONT should be rewritten from scratch.
- *No Unicode support.* TEXTRACE operates on normal TEX fonts with 256 or less glyphs. Although theoretically it wouldn't be hard to implement Omega and/or Unicode support, it is not available yet. TEXTRACE deals only with glyph indices, because encodings would be already resolved by the time fonts generated by TEXTRACE are needed.

PostScript font installation

Installing PostScript fonts for TEX in the general case is quite complicated. That's mostly because the font must be adapted to use a common TEX-specific encoding, accented glyphs should be composed by hand, kerning and ligature information must be adjusted etc. See the manual of DVIPS [7] or fontinst [5] for more information.

In the case of fonts generated by TEXTRACE the task is much easier. The metrics file (`.tfm`) requires no modifications, and moreover, it is already installed. Only the `.pfb` must be made available for TEX. TEXTRACE doesn't automate this process, the `.pfb` files must be installed manually. This consists of:

1. selecting the appropriate `.map` file to insert the font name and the filename. The file depends on the utility we'd like to use the font with:
   - *XDvi.* The file is `psfonts.map`, the location is probably `texmf/dvips/base`.
   - *DVIPS.* The file is any of the `.map` files mentioned in `config.ps`, most probably in `texmf/dvips/config`. `config.ps` almost always contains the entry for

`psfonts.map`.

- *pdfTEX.* The file is any of the `.map` files mentioned in `pdftex.cfg` (in `texmf/pdftex/config`). There is no default filename.

Note that the original `.map` files do not have to be overwritten, it is enough to place the new version into the current directory, thanks to Kpathsea.

2. appending a line describing the font to the `.map` file. This line is of the form:

   ⟨*TEX-font-name*⟩␣⟨*PostScript-FontName*⟩␣<⟨*PFB-file-name*⟩

   Example:

   ```
   ecrm1000 TeX-ecrm1000 <fcrm1000.pfb
   ```

   The ⟨*PostScript-FontName*⟩ can be read from the `.pfb` file after `/FontName`, but in the case of TEXTRACE it is simply the word `TeX-` prepended to ⟨*TEX-font-name*⟩. See the file `contrib/ecpfb.map` of the TEXTRACE distribution.

3. copying the `.pfb` file to its final, system-wide location. This step isn't mandatory, the font is also functional in the current directory, thanks to Kpathsea. The most common system-wide location for Type 1 fonts is `texmf/fonts/type1/*`.

4. testing whether the utility really finds the `.pfb` file by running it on a sample document and checking the console output whether it includes the `.pfb` file.

Other uses of TEXTRACE

Although TEXTRACE's primary goal is to convert METAFONT fonts to the Type 1 format automatically, the program is a bit more general: it can convert any font available to DVIPS (practically any TEX font) to Type 1, regardless of the original representation of the font.

The Type 1 format is standard since 1985, and TEXTRACE creates files that are compatible with all common font handling utilities (including many font editors, DTP programs, converters, printers, typesetters). Type 1 fonts can be easily converted to any of the font formats used nowadays. So TEXTRACE can be used as a first step (the TEX-specific step) to make a TEX font available in any publishing environment.

TEXTRACE converts only the glyph outlines, so kerning, ligature and other metrics information (all located in the `.tfm` file) must be converted separately. The documentation of TEXTRACE contains details about the automation of this process. Today's fonts tend to have more and more glyphs, exceeding TEX's limit of 256. It is common that a font is divided to smaller TEX fonts. These sub-fonts should be merged in a font editor after running TEXTRACE on all of them. There might be difficulties with kerning pairs and ligatures crossing font boundaries.

Alternatives

The method used by TEXTRACE isn't ultimate, there are many other possible solutions.

Respects of comparison

(F) *Final quality of the output.* This is the most important respect. Legibility and beauty must be considered in both low (screen) and high (printer) resolutions. The converted font must be accurately reflect the original, there should be no visual difference, not even in the smallest corners.

(C) *Compatibility.* The output must be compatible with both document preparation and reader software.

(S) *Size.* Although the capacity of hard disks increase rapidly nowadays, font files must be kept small because they might be loaded into printers very limited memory (even as few as 512 kb). The font sizes also affect download time of the PDF document.

(A) *Amount of human effort needed.* The more automated the process is, the better. Human effort is expected to be much for TEX fonts, because they are available in many (often 10 or more) design sizes and many styles.

Possible approaches

- Avoid conversion by using a virtual fonts to get a mixture of fonts that are already available in Type1 format. (See the subsection about EC fonts above.)

  The most serious limitation is that this approach is useless if no similar font is available in Type 1 format. (F), (C), (S): no problems. (A): it is not difficult to arrange that metrics and displacements in the virtual fonts are generated automatically.

  The AE, ZE and ZD fonts use this method.

- Design new fonts parallel in METAFONT and Type1. Develop a system that glues existing tools to aid parallel font development by allowing generation of both fonts from the same (new) source file (see [4]).

  The most serious problem is that existing fonts cannot be converted this way.

- Modify METAFONT somehow to output Type1 fonts (or vector outlines) instead of bitmaps.

  Modifying the `mf` engine itself would require too much effort. Another way is to modify only the output generation process of METAFONT by adding macros and `.log` file processing scripts. Such utilities already exist (for example: mf2ps by Daniel Berry and Shimon Yanai and the MF-PS package by Boguslav Jackowski), but they no longer work, cannot comply the restrictions of the Type 1 format or work only for `.mf` files specially written for using them.

- Interpret and execute `.mf` files directly, suppressing METAFONT altogether.

  This would be as hard as to completely rewrite METAFONT from scratch having Type 1 output capability in mind, because METAFONT is a full, complex programming language and most `.mf` files make use of this complexity. A faithful rewrite of METAFONT is really hard because there are too many (dirty) tricks in both the language and the current implementation.

- Post-process METAFONT's output.

  Effectively, post-processing means tracing. This is what TEXTRACE does. (F): font quality is far from the best, because of hinting information is missing. Traced

fonts looks almost always ugly in low resolutions, unless (H) a hinting is inserted manually, using a great deal of human resources. (S): font size can be reduced with sophisticated tracing techniques, possibly exploiting the curve representation model of the renderer. AutoTrace isn't so sophisticated, so it generates somewhat large fonts.

- Use METAPOST instead of METAFONT to interpret MF files. Post-process METAPOST's output.

  METAPOST is a METAFONT-clone with PostScript output. The PostScript it generates is suitable for conversion to Type 1 outlines.

  The commercial software MetaFog does this. It also makes the outline comply restrictions in Type 1. Such a restriction is that all paths must be filled, not stroked, and overlaps must be removed. It is theoretically possible to achieve this, but currently no robust implementation exists, not even MetaFog.

  There are fundamental problems with METAPOST itself. First, its language is only a subset of METAFONT's, so most fonts (including the EC fonts) don't even compile. Second, it cannot output strokes of elliptical or more complicated pens (which is required by some fonts). So it is a matter of luck whether one can succeed with METAPOST for a specific font.

- Apply a mixture of these methods.

  For example, one may get most glyphs from existing Type 1 fonts, and trace only the missing glyphs.

For the PDF problem

Our original goal was to find out a method for embedding TEX fonts into the PDF file so that the document would look nice and display quickly in PDF viewers, especially in Acrobat Reader.

It turned up that this can be ensured by inserting the fonts in some vector outline format. The quality of the font (minimal file size, presence of good hinting) seemed to be subsidiary when embeding it into PDF: current PDF viewers tend display even medium-quality fonts nicely. The conversion target must be undoubtly a vector outline format, but not necessarily Type 1. Another alternative is TrueType, which differs from Type 1 in two main respects:

- Hinting mechanism is much better, fonts are eye-pleasing to read even on screen and without antialiasing.
- It uses second order curves instead of third order (Bézier-) curves. Second level curves approximate the outline less precisely, so more control points are needed to achieve the quality of Type 1. This results larger file sizes.

For our intentions, neither of these respects is really relevant; the two font formats provide an equally good solution for the problem. However, considering compatibility issues, Type 1 turns out to be more adequate: it is older than TrueType, and more TEX-related utilities support it. Moreover, Type 1 has no dialects, can be represented in plain text canonically, and is easier to assemble and disassemble.

Conclusion

For those dealing with computers, Ugly Bits are experienced in everyday life, not just in memories. European TEX users have suffered for years because of an Ugly Bit in font inclusion to PDF documents. With the appearance of TEXTRACE, the problem became less severe (and documents became much less ugly).

Font conversion retaining the quality of the original is almost as tough as designing new fonts. For best results, both of them must be done manually. However, hand-converting hundreds of similar fonts is an enormous work, needs expert level knowledge and isn't really inspiring. Probably that's the reason why most of the TEX fonts weren't available in Type 1 format despite of the great need for it. (It is clear that the conversion in high quality is *possible*.)

TEXTRACE is an automatic and generic converter. It doesn't provide the best quality theoretically achievable in today's technology and human resources, but its output is good enough for most purposes. The three most important benefits of TEXTRACE are that it works for all TEX fonts (with a few technical exceptions), it runs completely automatically, without any human intervention, and it is free.

Due to the reasons mentioned above, professional quality Type 1 variants of all popular TEX fonts are not expected to appear for years. Until this happens, fonts generated by TEXTRACE will be the best alternative.

References

[1] Adobe Systems Inc. *Adobe Type 1 Font Format*. Addison–Wesley Publishing Company, 1995.

[2] Alexander Berdnikov, Yury Yarmola, Olga Lapko and Andrew Janishewsky. *Some experience in converting LH Fonts from METAFONT to Type1 format*. Proceedings of the 2000 Annual Meeting, TUGboat, 21(3), 2000.

[3] Taco Hoekwater. Generating Type 1 Fonts from Metafont Sources. *Proceedings of the TUG conference*, TUGboat 16(3), 1995.

[4] Boguslaw Jackowski, Janusz M. Nowacki and Piotr Strzelczyk. *METATYPE1: a METAPOST-based engine for generating Type 1 fonts*. Proceedings of the EuroTeX conference, 2001.

[5] Alan Jeffrey and Rowland McDonnell. *fontinst – Font installation software for TEX, fontinst v1.8*. `teTEX:doc/fontinst/fontinst.dvi.gz`, 1998.

[6] Hàn Thế Thành, Sebastian Rahtz and Hans Hagen. *The pdfTEX user manual*. `teTEX:doc/pdftex/base/pdftexman.pdf.gz`, 1999.

[7] Tomas Rokicki. *Dvips: A DVI-to-PostScript Translator*. `teTEX:doc/programs/dvips.dvi.gz`, 1997.

[8] George Williams. *PfaEdit – A PostScript® Font Editor, User's manual*. `http://pfaedit.sourceforge.net/overview.html`, 2000.

♢ ♢ ♢

Math typesetting in TEX: The good, the bad, the ugly

Ulrik Vieth

abstract. Taking the conference motto as a theme, this papers examines the good, the bad, and the ugly bits of TEX's math typesetting engine and the related topic of math fonts. Unlike previous discussions of math fonts, which have often focussed on glyph sets and font encodings, this paper concentrates on the technical requirements for math fonts, trying to clarify what makes implementing math fonts so difficult and what could or should be done about it.

keywords: math typesetting, math fonts, symbol fonts, font metrics, font encodings

Introduction

The topic of math typesetting (in general) and math fonts (in particular) has been a recurring theme at TEX conferences for many years. Most of the time these papers and talks have focussed on developing new math font encodings [1–3], updating and standardizing the set of math symbols in Unicode [4–6], or on implementing math fonts for use with a variety of font families [7–11]. However, fundamental technical issues and limitations of TEX's math typesetting engine have only rarely been addressed [12–15], usually in conjunction with a broader discussion of TEX's shortcomings.

In this paper we shall examine the latter topic in detail, trying to clarify what are the good, the bad, and the ugly bits of TEX's math typesetting engine.

Math typesetting: some good and some bad news

Let's start with the good news: Even after some twenty years of age, TEX is still very good at typesetting math. While some other systems such as Adobe InDesign have been catching up in the domain of text typesetting, even borrowing ideas from TEX's algorithms, math typesetting remains a domain where TEX is still at its best.

Whereas other systems usually tend to regard math as an add-on feature for a niche market that's very costly to develop and rarely pays off, math typesetting has always played a central role in TEX. In fact, math typesetting has been one of the main reasons why TEX was developed in the first place and why it has become so successful in the academic community and among math and science publishers.

While there are some subtle details that TEX can't handle automatically and that might benefit from a little manual fine-tuning, TEX usually does a very good job of typesetting properly-coded math formulas all by itself, without requiring users to care about how TEX's math typesetting engine actually works internally.

In general, an experienced TEX user, who has taken the time to learn a few rules and pay attention to a few details, can easily produce publication-quality output of even the most complicated math formulas by tastefully applying a few extra keystrokes to help TEX a little bit. And even an average TEX user, who is unaware of all the subtle details, can usually produce something good enough to get away with for use in seminar papers or thesis reports, often producing better result than what a casual user would get from so-called equation editors of typical word processors.

So, what's the bad news, after all? Actually, the problems only begin to emerge when leaving the user's perspective behind and looking at TEX's math typesetting engine from the implementor's point of view. While the quality of math typeset with TEX is probably still unmatched, some aspects of the underlying math typesetting engine itself are unfortunately far from perfect.

As anybody can tell, who has ever studied Appendix G of *The TEXbook*, trying to understand what's really going on when typesetting a math formula, TEX's math typesetting engine is a truly complicated beast, which relies on a number of peculiar assumptions about the way math fonts are expected to be built. Moreover, there are also some limitations and shortcomings where TEX begins to show its age.

Math typesetting: some technical background

Before we get into further details, it may be helpful to summarize how TEX's math mode differs from text mode and what goes on when typesetting text or math.

What goes on in text mode: `\char`*s, fonts and glyphs*

In text mode, when typesetting paragraphs of text, TEX essentially does nothing but translate input character codes to output codes using the currently selected font, assemble sequences of boxes and glue (i. e. letters or symbols represented by their font metrics and interword spaces) into paragraphs, and subsequently break paragraphs into lines and eventually lines into pages, as they are shipped out.

Whatever the levels of abstraction added by font selection schemes implemented in complex macro packages such as LATEX or CONTEXT, all typeset output is essentially generated by loading a particular font (i. e. a specific font shape of a specific family at a specific design size encoded in a specific font encoding) using the `\font\f=`⟨*font*⟩ primitive, selecting that font as the current font, and accessing glyphs from that font through the code positions of the output encoding.

Most input characters typed on the keyboard (except for those having special `\catcode`s for various reasons) are first translated to TEX's internal encoding (based on 7-bit ASCII and the `^^` notation for 8-bit codes), from which they are further translated to output codes by an identity mapping. (There is no such thing as a global

`\charcode` table to control this mapping.) Additional letters and symbols (such as `\ss` for 'ß') can be accessed through the `\char`⟨*code*⟩ primitive or by macros using `\chardef\c=`⟨*code*⟩, where ⟨*code*⟩ depends on the font encoding.

In actual fact, there are, of course, some further complications to typesetting text beyond this superficial description, such as dealing with ligatures, accented letters, or constructed symbols. Moreover, there are additional input methods than just converting characters typed on the keyboard or accessed through macros, such as using active characters or input ligatures to access special characters, but we don't want to go too far into such encoding issues in this paper.[1]

What goes on in math mode: `\mathchar`*s, math symbols and math families*

When it comes to math mode, things are, of course, a little more complicated than in text mode. For instance, TEX doesn't deal with specific fonts and character codes in math mode, but uses the concepts of math families and math codes instead. Whereas modern implementations of TEX provide room for several hundreds of text fonts, there is a limit of only 16 math families, each containing at most 256 letters or symbols. Compared to a text font, representing a specific font shape at a specific size, a math family represent a whole set of corresponding symbol fonts, which are loaded at three different sizes known as textstyle, scriptstyle and scriptscriptstyle.

In a typical setup of TEX, there should be at least four math families preloaded, where family 0 is a math roman font, family 1 is a math italic font, family 2 contains math symbols, and family 3 contains big operators and delimiters. Some assumptions about this are actually hard-wired into TEX, such as the requirement that the fonts used in families 2 and 3 have to provide a number of `\fontdimen` parameters controlling the placement of various elements of math formulas.

Any letter or symbol used in math mode, whether typed on the keyboard or accessed through a macro, is always represented by a math code, usually written as 4-digit hexadecimal number. In addition to specifying a math family and a character code, the math code also encodes information about the type of a symbol, whether it is an ordinary symbol, a big operator (such as $\int$), a binary operator (such as $+$), a relation (such as $=$), an opening or closing delimiter, or a punctuation character. (There is also a special type of ordinary symbols, which are allowed to switch math families. This particular type is mostly used for alphabetic symbols.)

The mapping of input characters typed on the keyboard to corresponding symbols is controlled through a `\mathcode` table, which by default maps letters to math italics and numbers to math roman. Additional math symbols including the letters of the greek alphabet can be accessed by macros using `\mathchardef\c=`⟨*code*⟩, where ⟨*code*⟩ is a math code composed of type, math family and character code. In a similar way, special types of symbols such as delimiters and radicals are handled using macros involving `\delimiter`⟨*code*⟩ or `\radical`⟨*code*⟩.

[1] LATEX uses the `inputenc` and `fontenc` packages to deal with 8-bit input and output encodings beyond 7-bit ASCII. Most 8-bit input codes for accented letters are first mapped to replacement macros through active characters. These, in turn, are subsequently mapped back to 8-bit output codes. For a detailed discussion on what really goes on internally in the various processing stages and what constitutes the subtle differences between characters, glyphs, and slots, see [16].

Considering the two-dimensional nature of typesetting math, it should be obvious that there is much more to it than simply translating input math codes to output character codes of specific fonts. In addition to choosing the proper symbols (based on the math families and character codes stored in the math codes), it is equally important to determine the proper size (based on the three sizes of fonts loaded in each math family) and to place the symbols at the proper position relative to other symbols with an appropriate amount of space in between. Here, the type information stored in the math codes comes into play, as TEX uses a built-in spacing table to determine which amount of space (i. e. either a thin space, medium space, thick space, or no space at all) will be inserted between adjacent symbols.

Interaction between the math typesetting engine and math fonts

It is interesting to note that TEX's math typesetting engine relies on a certain amount of cooperation between its built-in rules, parameters set up in the format file, and parameters stored in the font metrics of math fonts.

For example, when determining the spacing between symbols, the spacing table that defines which amount space will be inserted is hard-wired into TEX, while the amounts of space are determined by parameters such as `\thinmuskip`, `\medmuskip` or `\thickmuskip`, which are set up in the format file. These parameters are defined in multiples of the unit 1 mu = 1/18 em, which, in turn, depends on the font size. Similarly, when processing complex sub-formulas, such as building fractions, attaching subscripts and superscripts, or attaching limits to big operators, the actual typesetting rules are, of course, built into TEX itself, whereas various parameters controlling the exact placement are taken from `\fontdimen` parameters.

In view of the topic of this paper, it should be no surprise that such kind of close cooperation between the math typesetting engine and the math fonts does not come without problems. While there are good reasons why some of these parameters depend on the font metrics, it might be a problem that their scope is not limited to the individual fonts loaded in math families 2 and 3; they automatically apply to the whole set of math fonts. (This is usually not a problem when a consistent set of math fonts is used, but this assumption might break and might lead to problems when trying to mix and match letters and symbols from different sets of math fonts.)

Specific problems of TEX's math fonts

After reviewing the technical background of math typesetting, we shall now look into some specific problems of TEX's math typesetting engine. In particular, we will focus on those problems that make it hard to implement new math fonts.

Glyph metrics of ordinary symbols: When the TFM *width isn't the real width ...*

Perhaps the most irritating feature of TEX's math fonts is the counter-intuitive way, in which glyph metrics are represented differently from those of text fonts. Normally, the font metrics stored in TFM files contain four fields of per-glyph information for each character or symbol: a *height* (above the baseline), a *depth* (below the baseline),

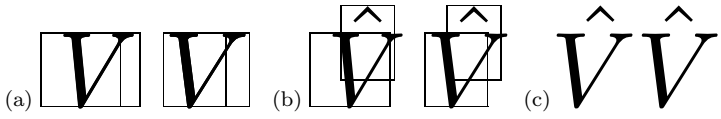


FIGURE 1: PLACEMENT OF ACCENTS IN TEXT MODE AND MATH MODE, COMPARING `cmti10` AND `cmmi10` AT 40 PT. (A) COMPARING THE GLYPH WIDTHS AND SIDE-BEARINGS FOR TEXT ITALIC AND MATH ITALIC. (B) COMPARING THE RESULTS OF TEX'S DEFAULT ALGORITHM FOR ACCENT PLACEMENT, PRODUCING SLIGHTLY DIFFERENT RESULTS FOR SLIGHTLY DIFFERENT METRICS OF TEXT ITALIC AND MATH ITALIC. (C) COMPARING THE RESULTS OF TEX'S `\accent` AND `\mathaccent` PRIMITIVES, ILLUSTRATING THE CORRECTION DUE TO `\skewchar` KERNING.

a *width*, and an *italic correction* (which might be zero for upright fonts). In math fonts, however, glyph metrics are interpreted differently. Since additional information needs to be stored within the framework of the same four fields of per-glyph information, some fields are interpreted in an unusual way: The *width* field is used to denote the position where subscripts are to be attached, while the *italic correction* field is used to denote the offset between the subscript and superscript position. As a result, the real width isn't directly accessible and can only be determined by adding up the *width* and *italic correction* fields. Moreover, the information stored in the *width* field usually differs from the real width, which causes subsequent problems.

Most importantly, this peculiar representation of glyph metrics causes a lot of extra work for implementors of math fonts, since they can't simply take an italic text font and combine it with a suitable symbol font to make up a math font. Instead the metrics taken from an italic text font have to be tuned by a process of trial and error and subsequent refinements to arrive at optimal values for the placement of subscripts and superscripts as well as for the side-bearings of letters and symbols.

Placement of math accents: When you need a `\skewchar` *to get it right ...*

Another problem related to glyph metrics arises as an immediate consequence of the previous one. Since the *width* field of the glyph metrics of math fonts doesn't contain the real glyph width, TEX's default algorithm for placing and centering accents or math accents doesn't work, and a somewhat cumbersome work-around was invented, the so-called `\skewchar` mechanism. The basic idea is to store the shift amounts to correct the placement of math accents in a set of special kern pairs in the font metrics. To this effect, a single character of each math font (usually a non-letter) is designated as the `\skewchar` and kern pairs are registered between all other characters that may be accented (letters or letter-like symbols) and the selected `\skewchar`.

As in the previous case, the most important problem of the `\skewchar` mechanism (apart from being hack) is that it causes extra work to implementors of math fonts. Instead of being able to rely on TEX's default algorithm for the placement of accents, the `\skewchar` kern pairs have to be tuned to arrive at optimal values. Moreover, the choice of the `\skewchar` has to be considered carefully to avoid interference with normal kern pairs in math fonts, such as between sequences of ordinary symbols or between ordinary symbols and delimiters or punctuation.

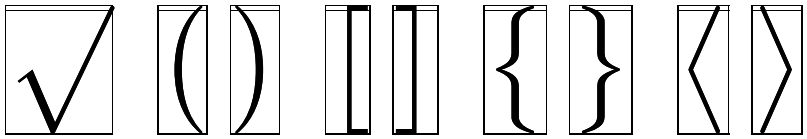


FIGURE 2: GLYPH METRICS OF BIG RADICALS AND DELIMITERS IN MATH EXTENSION FONTS (SHOWING `cmex10` AT 40 PT). THE HEIGHT ABOVE THE BASELINE MATCHES EXACTLY THE DEFAULT RULE THICKNESS, AS REQUIRED FOR THE RULE PART OF RADICALS. ALL GLYPHS OF THE SAME SIZE ARE PLACED IN THE SAME POSITION TO COPE WITH LIMITATIONS OF THE TFM FILE FORMAT.

(Another problem related to kerning in math fonts is that TEX doesn't support kerning between multiple fonts, so it isn't possible to define kern pairs between upright and italic letters taken from different fonts, but that's another story.)

Glyph metrics of big symbols: When glyphs hang below the baseline ...

Another quite different problem related to glyph metrics, which occurs only in the math extension font, is the placement of big symbols (big operators, big delimiters and radicals) within their bounding boxes. As anyone will have noticed, who has ever printed out a font table of `cmex10` using `testfont.tex` or has looked at Volume E of *Computers & Typesetting*, most symbols in the math extension font have very unusual glyph metrics, where the glyphs tend to hang far below the baseline.

The reasons for this are a little complicated and quite involved. To some extent they are due to technical requirements, such as in the case of big radicals where the height above the baseline is used to determine the rule thickness of the horizontal rule on top of a root. However, in other cases, such as big operators and delimiters, there are no technical requirements for the unusual glyph metrics (at least not in TEX82) and the reasons are either coincidental or due to limitations of the TFM file format, which doesn't support more than 16 different heights and depths in one font.

(Incidently, the height of big radical glyphs is usually exactly the same as the default rule thickness specified in the `\fontdimen` parameters, so one could have used just that instead of relying on the glyph metrics to convey this information.)

What is particularly irritating about this problem is that math fonts featuring such glyph metrics are usually not usable with any other typesetting system besides TEX. While TEX automatically centers operators, delimiters and radicals on the math axis, most other systems expect to find glyphs in an on-axis position as determined by the type designer. It therefore becomes extremely hard, if not impossible, to develop math fonts equally usable with TEX and with other math typesetting systems. (The font set distributed with *Mathematica* avoids this problem by providing two different sets of radicals, occupying different code positions in the font encoding.)

Extensible delimiters: When the intelligence lies in the font ...

Speaking of math extension fonts, there is another issue related to the way intelligence and knowledge is distributed between math fonts and TEX itself. As was mentioned before, TEX uses so-called math codes to represent all kinds of math symbols, encoding

a type, a family and a code position in a 4-digit hexadecimal number. Depending on the type, however, this might not be everything to it, as further information might also be hidden in the font metrics of math fonts.

While ordinary symbols are represented by a single glyph in the output encoding, big operators usually come in two sizes known as textstyle and displaystyle. However, TEX's macro processor only uses a single math code (and hence, only a single code position) to represent the smaller version of a big operator, while it is left to the font metrics of the relevant math font to establish a link to the bigger version through a so-called charlist in the font metrics. (This kind of font information is, of course, also accessible to TEX's typesetting engine, but not to the macro processor.)

In a similar way, the big versions of delimiters and radicals also rely on the font metrics to establish a chain of pointers to increasingly bigger versions of the same glyph linked through a charlist. Additionally, the last entry of the chain might represent an entry point to a so-called extensible recipe, referencing the various building blocks needed to construct an arbitrarily large version of the requested symbol.

What is extremely confusing about this, is that the code positions used to access extensible recipes could be completely unrelated to the actual content of these slots. In some cases, they might be chosen carefully, so that the slots used as entry points are the same as those containing the relevant building blocks. In other cases, however, an entry point might be chosen simply because it isn't already used for anything else, but it might actually refer to glyphs taken from completely different slots.

Limitations and missing features of the math typesetting engine

So far, we have looked at some specific problems that are often brought up when discussing the difficulties of implementing new math fonts for TEX. While TEX works perfectly well as it is currently implemented, some of these very peculiar features may well be considered examples of bad or ugly design that are worth reconsidering. Apart from that, there are also some limitations as to what TEX's math typesetting engine can do and what it can't do. Therefore, there is also some food for thought regarding additional features that might be worth adding in a successor to TEX.

Size scaling and extra sizes in Russian typographical traditions

As explained in detail in Appendix G of *The TEXbook*, the functionality of TEX's math typesetting engine is based on a relatively small number of basic operations, such as attaching subscripts and superscripts, applying and centering math accents, building fractions, setting big operators and attaching limits, etc. In these basic operations, TEX relies on some underlying concepts of size, such as that there are four basic styles known as displaystyle, textstyle, scriptstyle and scriptscriptstyle, which are chosen according to built-in typesetting rules that can't be changed.

As was pointed out in [17], however, these built-in typesetting rules and the underlying concepts of size might not really be sufficient to cover everything needed when it comes to dealing with specific requirements for traditional Russian math typesetting, which has quite different rules than what is built into TEX.

While TEX only supports two sizes of big operators in textstyle and displaystyle, Russian typography requires an additional bigger version (as well as an extensible version of a straight integral) for use with really big expressions. Similarly, while TEX essentially uses only three sizes to go from textstyle to scriptstyle and scriptscriptstyle in numerators and denominators of fractions or in subscripts and superscripts, Russian typography calls for another intermediate step, making it necessary to have a real distinction between the font sizes used in displaystyle and in textstyle.

Extensible wide accents and over- and underbraces

While changes to fundamental concepts such as the range of sizes in math mode would have far-reaching consequences that are very difficult to assess and to decide upon, there are other potentially interesting features that might be easier to implement, even within the framework of the existing TFM file format.

One such example would be extensible versions of wide accents, which might also be used to implement over- and underbraces in a more natural way. The reason why this would be possible is simply that the TFM file format supports charlist entries and extensible recipes for any glyph. It only depends on the context whether or not these items are looked at and taken into account by TEX. In the case of delimiters and radicals, TEX supports a series of increasingly bigger versions linked through a charlist as well as an extensible recipe for a vertically extensible version. In the case of wide accents, however, TEX only supports a series of increasingly wider versions linked through a charlist, but no extensible recipe for a horizontally extensible version, even if the font metrics would support that.

Given a new mechanism for horizontally extensible objects similar to the existing mechanism for vertically extensible delimiters, it would also be possible to reimplement over- and underbraces in a more natural way, without having to rely on complicated macros for that purpose. (The font set distributed with *Mathematica* already contains glyphs for over- and underbraces in several sizes as well as the building blocks for extensible versions. Moreover, the *Mathematica* font set also contains similar glyphs for horizontally extensible versions of parentheses, square brackets and angle brackets, which don't exist in any other font set.)

Under accents, left subscripts and superscripts

Two other examples of potentially interesting new features would be mechanisms for under accents and for left subscripts and superscripts. While support for under accents might be feasible to implement given that over accents are a special type of node in TEX's internal data structures anyway, adding support for left subscripts and superscripts would certainly be more complicated, considering that right subscripts and superscripts are an inherent feature of all types of math nodes.

As for an implementation of under accents in the framework of the existing TFM file format, it would probably be necessary to resort to another cumbersome workaround similar to the `\skewchar` mechanism in order to store the necessary offset information. A macro solution for under accents that uses reversed `\skewchar` kern pairs has already been developed in the context of experimental new math font encodings [2].

Summary and Conclusions

What are the reasons for all these problems?

It is pretty obvious that most of the problems of math fonts discussed in this paper can be traced back to the time when TEX was developed more than twenty years ago. Given the scarcity and cost of computing power, memory and disk space at that time (in the late 1970s and early 1980s), it is no surprise that file formats such as TFM files for font metrics were designed to be compact and efficient, providing only a limited number of fields per glyph and a limited number of entries in lookup tables.

Based on such a framework, compromises and workarounds such as overloading some fields in math fonts to store additional information were unavoidable, even though such hacks damaged the clarity of design and eventually lead to other problems, requiring even further hacks to deal with the consequences (such as the `\skewchar` mechanism to compensate for the fact that the TFM width didn't represent the real glyph width). In view of this, it is no surprise that overcoming limitations (such as being limited to 16 math families or 16 TFM heights and depths) is the highest priority on the wish list before cleaning up other problems or adding new features.

What's good, what's bad, what's ugly?

Speaking of good, bad and ugly bits, the conference motto suggests: *"First of all, keep up the good bits and extend them if possible. Analyze the ugly bits, learn from them, and find easy and generic ways to get around them. Finally, find the bad bits and eradicate them!"* By these standards most of the problematic features discussed in this paper can probably be classified as ugly bits, with very few exceptions that might also be considered bad bits, whereas some (but not all) of the suggested new features could be summarized as extending the good bits.

As for extending the good bits, adding extensible versions of wide accents or support for under accents might be feasible examples, that could be implemented relatively easily, whereas other suggested new features such as adding support for left subscripts and superscripts or introducing additional sizes might have far-reaching consequences that should be considered with care, so as not to introduce new problems.

As for eradicating the bad bits, reconsidering the algorithm for typesetting radicals might be a high priority item on the wish list. As suggested by [13], using a repeated glyph for the rule part instead of a horizontal rule whose height depends on the glyph metrics might be a feasible solution for a better implementation.

As for learning from the ugly bits and finding better ways to get around them, starting over with a completely new font metrics format as suggested in [15] to overcome the current limitations would certainly help to avoid most of the remaining problems. Given that compactness of file formats and efficiency of store are no longer real issues with modern computers, it would be no problem to use a human-readable verbose file format and to extend the font metrics by any number of additional fields as needed to convey additional information. This way, many problems caused by overloading certain fields of the glyph metrics or resorting to workarounds such as the `\skewchar` mechanism could all be avoided. Considering that, there is hope that dealing with math fonts could eventually become much easier than it is today!!!

REFERENCES

[1] Alan Jeffrey. Math font encodings: A workshop summary. *TUGboat*, 14(3):293–295, 1993.

[2] Matthias Clasen and Ulrik Vieth. Towards a new Math Font Encoding for (LA)TEX. *Cahiers GUTenberg*, 28–29:94–121, 1998. Proceedings of the 10th European TEX Conference, St. Malo, France, March 1998.

[3] Ulrik Vieth. Summary of math font-related activities at EuroTEX '98. *MAPS*, 20:243–246, 1998.

[4] Taco Hoekwater. An Extended Maths Font Set for Processing MathML. In *EuroTEX'99 Proceedings*, pages 155–164, 1999. Proceedings of the 11th European TEX Conference, Heidelberg, Germany, September 1999.

[5] Patrick Ion. MathML: A key to math on the Web. *TUGboat*, 20(3):167–175, 1999.

[6] Barbara Beeton. Unicode and math, a combination whose time has come – Finally! *TUGboat*, 21(3):176–186, 2000.

[7] Alan Jeffrey. PostScript font support in LATEX2ε. *TUGboat*, 15(3):263–268, 1994.

[8] Thierry Bouche. Diversity in math fonts. *TUGboat*, 19(2):121–134, 1998.

[9] Alan Hoenig. Alternatives to Computer Modern Mathematics. *TUGboat*, 19(2):176–187, 1998.

[10] Alan Hoenig. Alternatives to Computer Modern Mathematics. *TUGboat*, 20(3):282–289, 1999.

[11] Richard J. Kinch. Belleek: A call for METAFONT revival. *TUGboat*, 19(3):244–249, 1998.

[12] Berthold Horn. Where are the math fonts? *TUGboat*, 14(3):282–284, 1993.

[13] Matthias Clasen. Ideas for improvements to TEX's math typesetting in ε-TEX/NTS. unpublished paper, available from `http://www.latex-project.org/papers/etex-math-notes.pdf`, 1998.

[14] David Carlisle. Notes on the Oldenburg ε-TEX/LATEX3/CONTEXT meeting. unpublished paper, available form `http://www.latex-project.org/papers/etex-meeting-notes.pdf`, 1998.

[15] NTG TEX Future Working Group. TEX in 2003: Part I: Introduction and Views on Current Work. *TUGboat*, 19(3):323–329, 1998.

[16] Lars Hellström. Writing ETX format font encoding specifications. unpublished paper, available from `http://abel.math.umu.se/~lars/encodings/encspecs.tex`, 2001.

[17] Alexander Berdnikov. Russian Typographical Traditions in Mathematical Literature. In *EuroTEX'99 Proceedings*, pages 211–225, 1999. Proceedings of the 11th European TEX Conference, Heidelberg, Germany, September 1999.

♢ ♢ ♢

'Typography' and production of manuscripts and incunabula

Paul Wackers

ABSTRACT. This paper describes how the modern type of books slowly came into existence during the middle ages. The first printers modeled their products on these handwritten books but needed – in time – some adjustments because of the differences in production between a manuscript and a printed book and because of the differences between producing for a patron or for an anonymous mass market.

KEYWORDS: Typography, manuscript, incunabulum

When we try to produce well-structured books that are also pleasing to the eye, we stand in a tradition of more than twenty centuries. The appearance of modern western books, however, slowly developed during the middle ages and got its definitive form in the decades round 1500, the first phase of book printing in Europe. Most classical books had the form of scrolls but 'our' type of books, the *codex* form, came into existence before the beginning of the fifth century. The typographical conventions we follow, however, developed during the middle ages.

The Romans had two types of script: a 'capital' font used for inscriptions and formal writing, and a 'cursive' font for personal use. The second type became the book script of the early middle ages. In the seventh century, probably first on the British Isles, these two were combined. One used single capital letters in the 'cursive' script to mark the beginning of sentences. This is the beginning of the use of capitals and lower case in texts. Somewhat later the usage came into existence to distinguish words by placing blanks in between. (In classical books all letters form one big mass.) At the same time also the first punctuation signs were developed.

Capitals are used to distinguish sentences from each other. But most texts need also 'higher' structures. To mark these 'sections' initials were developed, again in the seventh century. Initials are larger letters (two, three, sometimes even more lines high) which have often a different color and some decoration. Because these initials could not be placed in the middle of normal text, they always came at the beginning of a text line or in the margin. Thus also the idea of paragraphs developed.

This principle was gradually expanded. One got used to a hierarchy of initials and other signs to mark specific elements of texts (e.g. paragraph signs: ¶)

When the need arose to add information to the main text which was related to it but not part of it (e.g. explanatory notes) this happened in smaller script, sometimes

between the lines of the main text but often in the margins. These 'notes' are called *glosses*. This is the beginning of our system of (foot)notes. The links between main text and glosses was indicated by signs or by repeating some words from the main text (the *lemma*) in the gloss.

In the early middle ages most books had pages with only one column of text on each page (as in most of our books) but in the later middle ages most books had two columns. This was caused by a wish to fill the pages as much as possible. The most easily produced format of parchment was so wide that if one filled its width wholely with only one line, this made reading difficult. Hence the system with the two smaller columns was developed. This combined an optimum of text on a page with a good readability.

The first printed books, normally called *incunabula*, were produced to look like hand written books. After all everyone was used to that type of book. Because printing could only be done in one color, this implied that the printed pages needed to be completed manually afterwards. Initials, paragraph signs, chapter headings in red, etc. were separately added in most copies.

This last part of the production of printed books was the most expensive part, so printers sought ways to reduce the costs. Woodcuts were introduced als illustrations instead of miniatures and also woodcut initials were developed that could be used during the printing process instead of being added afterwards. Within a hunderd years the manually completion of printed books disappeared.

A major difference between a manuscript and a printed book is that the first is always a unique object and the second must be produced in large numbers to be economically successful. The owner of a manuscript knows what he possesses, so he does not need an indication of the contents. The potential buyer of a printed book does not have this information but has need of it. To fill this need first colophons were printed which gave some information about content and printer, but soon a title page was developed which offered information about the book and which functioned also as advertising.

During the conference these changes in lay out and production will be illustrated by means of a series of images, from a fifth century Virgil manuscript to sixteenth century fable collections. Special attention will be given to Gutenberg, the first printer in European history.

REINTRODUCING TYPE 3 FONTS TO THE WORLD OF TEX

Włodek Bzyl
matwb@univ.gda.pl

Abstract

Nowadays, a great number of documents are produced every day. Many authors would like their documents to stand out from the rest not only by content but also by typographic design. For this purpose one may use decorative letters, ornaments, dingbats and special fonts. If each document would have to look different from all the others a great many fonts and font deviations are needed. This could be achieved by combining the METAPOST language with the type 3 font format. This new font creation technology enables users endless single-use-only variations in weight and width, style and size, and in color. A new level of control over the embellishment level of fonts in documents is thereby achieved.

INTRODUCTION

For the past five centuries type has been cut in wood and cast in metal. The idea that a computer program could design type where letterforms are represented by procedures which draw each letter with lines and beziér curves has appeared recently. More than twenty centuries of build-up knowledge about geometry and curves proved to be applicable in this transition.

In 1977 the first METAFONT fonts were created by D. E. Knuth. In 1984 the first PostScript Type 1 fonts were created by Adobe Systems.

In Knuth's approach, shapes are designed in a declarative manner and are drawn with 'simulated pens'. In other words, the relationships which convey the idea of the design are encapsulated in a set of parameters and described in the language of algebra. Computer has to figure how to satisfy those conditions and produce the digitized image. For example, we could state that one curve should touch another in the middle, or that a pair of curves should cross at right angle. The Adobe approach is simpler. Shapes are described in an imperative manner. The outline of the letter is desribed by a series of curves and this outline is filled with 'ink'. Although Knuth's programs allow to generate endless variations of shapes, the world wide standard become Adobe's Type 1 outline font system, possibly because it is much easier to draw something than to explain how to draw it.

The PostScript language contains another method for defining a font named Type 3. It employs almost all the usual PostScript operators including color. Color inclusion could be vieved as an important extension to the old metal technology where letters are printed in ink color. Unfortunately current versions of PostScript do not cache color or grey images, so they are executed each time they are printed. This could slow down printing considerably. Maybe this extra added inefficiency made Type 3 fonts so rare species. But, even with today technology, there are areas where these fonts could make printed texts more readable, personalized and attractive. These include [2, p. 8–9]:

- Display fonts — designed to catch the eye. Used in titles, headings, posters, signs and adverbs.
- Decorative fonts — designed to convey a statement or a particular mood. Their choice depends on the job at hand. They are very susceptible to the vagaries of fashion. These include: initial caps, ornaments, tilings, emotional (smileys), logos.
- Speciality fonts — designed for particular purposes. Areas catered for include: phonetic symbols, mathematical operators, musical notation, dingbats and various oddities.

In 1989 the METAPOST system appeared. The author, J. D. Hobby, realised the power of Knuth's approach and its weakness in outputing black & white bitmaps. So he created a system which implements a picture drawing language like Knuth's METAFONT except it outputs PostScript commands instead of bitmaps. Because Type 3 fonts have a very simple format it is possible to postprocess METAPOST output and to create a Type 3 font.

I start with a quick glance into the past to show the use of special fonts in old books. Then, back to the future. In this section, I would like to draw your attention to several examples of Type 3 fonts usage. In the next two sections, I describe how to create a Type 3 font with METAPOST and how easy is to open Pandora's fonts box. There I propose a way of controling of multitude derived fonts. In appendices, I present a detailed description of Type 3 font format and basic METAPOST library supporting development of Type 3 fonts.

TYPOGRAPHICAL JOURNEY

The first two figures are taken from the books published by the Kelmscott Press. This publishing house was established and run by William Morris. Letterforms are derived from his own studies in printing history. The letterforms are chosen for their decorative quality. The third figure is taken from "The Art of Illuminating" by Matthew Dingby Wyatt and W. R. Tymms. This book described the pioneering efforts of its authors in the research of letterforms and manuscript illumination.

Note the use of ornaments and initial caps in Fig. 1. There is no 'bézierness' in their shapes, so I can hardly imagine to program them. In Fig. 2, in the text at

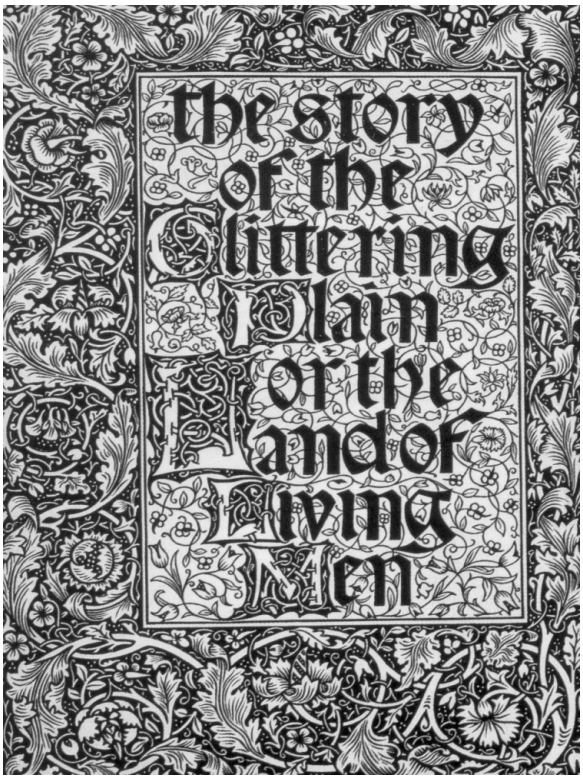


Fig. 1. Kelmscott Press: *The Story*, 1891

the bottom, dingbats are used to fill unwanted white space. Similar shapes can be found in contemporary computer fonts. The letter 'P' in Fig. 3 was hand colored. I think that existing computer tools cannot make creating such a beauty possible. Knuth [8] writes: "I do not believe that mathematical methods will resolve all the problems of digital typography; but I do believe that mathematics will help." This should be no suprise. It will always be possible to create shapes with a brush or chisel for which there is no mathematical description of their elegance and magic. But I do believe that new font technologies will make typographers and programmers collaborate eventually in a similiar manner as today designers and architects do.

So, let's go on and explore what new computers have to offer.

BACK TO THE FUTURE

Colored fonts. In traditional typography fonts could not be multicolored. "Poor men" substitutes for coloring is overlaying letters with patterns or using color inks. With Type 3 fonts created in METAPOST, shape, size, sidebearings, weight and colors could be adjusted to match surrounding text.

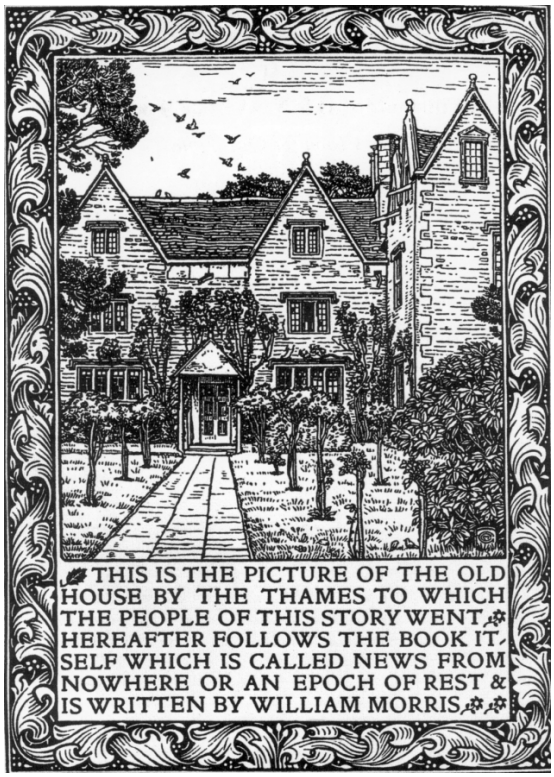


Fig. 2. Frontispiece: *News form Nowhere*, 1892

XI Computer Typography

1. Introduction to MetaPost
2. Constructions with compass and ruler
3. Introduction to TEX
4. Typography basics
5. Computer fonts

Fig. 3. M. D. Wyatt. The Art of Illuminating

Special symbols. In [11, p. v] Knuth writes: "In order to make it possible for many types of users to read this manual effectively, a special sign is used to designate material that is for wizards only: When the symbol

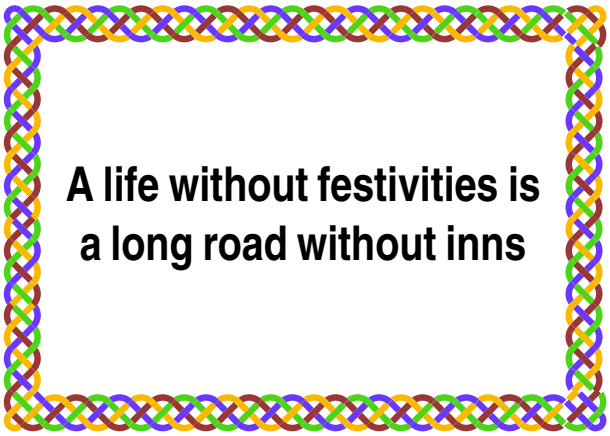
appears at the beginning of a paragraph, it warns of a 'dangerous bend' in the train of thought; don't read the paragraph unless you need to." This idea calls for more special symbols. Here are a few more examples [18, 19, 10].

No man can be a pure specialist without being in the strict sense an idiot.

Were we faultless, we would not derive such satisfaction from remarking the faults of others.

Type design can be hazardous to your other interests. Once you get hooked, you will develop intense feelings about letterforms; the medium will intrude on the messages that you read. And you will perpetually be thinking of improvements to the fonts that you see everywhere, especially those of your own design.

Frames. In [5] Gibbons writes: "Sadly, there is a shortage of good symbols for creating such ornaments; not many typographic elements come in eight different orientations! However, there is nothing to stop you designing your own symbols." You can not agree with that statement, can you [19].



Various oddities Math is plenty difficult in normal type. Programmers realized that their programs are easier to comprehend when typeset in color. So, what about coloring math formulas?

$$\sum_{a<k<b} f(k) \approx \int_a^b f(x)\,dx + \sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(x)\Big|_a^b$$

$$\binom{n}{k} \equiv \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor}\binom{n \bmod p}{k \bmod p} \pmod p$$

$$2 \uparrow\uparrow k \stackrel{\text{def}}{=} 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}k$$

$$M = \begin{array}{c} \\ C \\ I \\ C' \end{array} \begin{array}{c} \begin{array}{ccc} C & I & C' \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ b & 1-b & 0 \\ 0 & a & 1-a \end{pmatrix} \end{array}$$

$$k = 1.38 \times 10^{-16}\ \text{erg}/^\circ\text{K}$$

$$\tilde{\Phi} \subset NL_1^*/N = \tilde{L}_1^* \subseteq \cdots \subseteq NL_n^*/N = \tilde{L}_n^*$$

$$\sin 18^\circ = \frac{1}{4}(\sqrt{5}-1)$$

Or about having your own signature and stamps:

In this article I have used more Type 3 fonts than was really necessary, so it is high time to show how to create one.

DEVELOPING TYPE 3 FONT

Fonts are collections of shapes. A computer font is prepared in the form of a computer program. There are several kinds of fonts. Each type of font has its own convention for organizing and representing the information within it. The PostScript language defines the following types of fonts [3, p. 322]: 0, 1, 2, 3, 9, 10, 11, 14, 32, 42. Text fonts are mostly of Type 1. They are programmed with the special procedures. To execute efficiently and to produce more legible output, these procedures, use features common to collection of black & white letter-like shapes. They may not be used outside Type 1 font. While any graphics symbol may be programmed as a character in a Type 1 font, non-letter shaped symbols are better served by the Type 3 font program which defines shapes with ordinary PostScript procedures including these which produce color. Other font types are used infrequently.

Although Type 3 fonts are PostScript programs I prefer to program shapes in the METAPOST language and convert `mpost` output into Type 3 font, because the METAPOST language simplifies the programming due to its declarative nature. In PostScript each curve is build from lines, arcs of circle and beziér curves. For complicated shapes this requires a lot of nontrivial programming. METAPOST implements 'a magic recipe' [10] for joining points in a pleasing way. This helps a lot. Even if you are not satisfied with the shape, you can give the program various hints about what you have in mind, therefore improving upon automatically generated curve.

To use a font with TEX the font metric file is required. It contains data about width, height and depth of each shape from the font. Because `mpost` could generate metric file on demand, fonts prepared with METAPOST are immediately usable with TEX.

Creation of a Type 3 font is multi step process.

1. A font must be imagined and designed.
2. It must be programmed. METAPOST does not support that, but a specially created library of procedures does.
3. The program must be compiled.
4. The files thus created must be assembled into a font. This task is done by a PERL program.

Additionally, the font must be made available to TEX and instructions must be given to tell TEX to switch to this font.

METAPOST itself does not support font creation. So I have written a special `type3` library. It provides very basic routines for font creation. These include macros for glyph and font administration, macros for annotations of hardcopy proofs, and finally, macros which helps in the process of converting separate glyphs into a font. The convertion is done by a PERL program named `mptot3`. This program was designed after MF2PT3 tool [16] that generates a Type 3 font that correspond to a METAFONT program.

Here is an example.

Let us create a font which contain one character — plus. Use an ascii text editor, it does not have to be your favorite — any such editor works, to create a file called `plus-000.mp` that contains the following lines of text.

Each font program should name the font it creates.

```
font_name "Plus-000";
```

These names are merely comments which help to understand large collections of PostScript fonts.

```
family_name "Plus";
font_version "0.0final";
is_fixed_pitch true;
```

and following names play similiar rôle in the TEX world.

```
font_identifier:="PLUS 000";
font_coding_scheme:="FONT SPECIFIC";
```

The `mpost` program does all its drawing on its internal 'graph paper'. We establish 100×100 coordinate space on it.

```
grid_size:=100;
```

The font matrix array is used to map all glyphs to 1×1 coordinate space. This PostScript convention allows consistent scaling of characters which come from different fonts.

```
font_matrix
  (1/grid_size,0,0,1/grid_size,0,0);
```

This particular font matrix will scale a plus shape by the factor 1/100 in the x dimensions and by the same factor in the y dimension. If we had choosen scaling by the factor 1/50 then plus shape would have appeared twice bigger comparing to characters from other fonts.

The data below provides information about how to typeset with this font. A font quad is the unit of measure that a TEX user calls one '`em`' when this font is selected. The normal space, stretch, and shrink parameters define the interword spacing when text is being typeset in this font. A font like this is hardly ever used to typeset anything apart from the plus, but the spacing parameters have been included just in case somebody wants to typeset several pluses separated by quads or spaces.

```
font_quad:=100;
font_normal_space:=33;
font_normal_stretch:=17;
font_normal_shrink:=11;
```

Another, more or less, ad hoc unit of measure is `x_height`. In TEX this unit is available under the name '`ex`'. It it used for vertical measurements that depend on the current font, for example for accent positioning.

```
font_x_height:=100;
```

The plus font is an example of a parametrized font. A single program like this could be used to produce infinite variations of one design. For example, by changing the parameters below we could make the plus character to paint in different color, change width or change the stem width.

```
color stem_color;
stem_color:=red;
u:=1; % unit width
stem_width:=10;
```

The `mode_setup` macro could be used to override all the settings done above. Typically, it is used to tell the `mpost` program to generate a font metric file or proofsheets. Additionaly, `mode_setup` could execute any piece of valid METAPOST code at this point. For example, we could change the stem color to blue and the stem width to 5 units. The code to be executed could be read from a separate file (see the next section how to prepare and use such a file). Thus we can make a variation of this design or *re-parameterize* the font without changing the master `plus-000.mp` file. Such a mechanism is required. Otherwise, we populate our hard disks with similiar files.

```
mode_setup;
```

Type3 library makes it convenient to define glyphs by starting each one with:

beginpic (⟨code⟩, ⟨width⟩, ⟨height⟩, ⟨depth⟩)

where ⟨code⟩ is either a quoted single character like `"+"` or a number that represents the glyph position in the font. The other three numbers say how big the glyph bounding box is. The `endpic` finishes the plus glyph.

Each `beginpic` operation assigns values to special variables called `w`, `h`, and `d`, which represent respective width, height, and depth of the current glyph bounding box. Other pseudo-words are part of METAPOST language and are explained in [6].

```
beginpic("+",100u,100,0); "+  plus";
  interim linecap:=butt;
  drawoptions(withcolor stem_color);
  pickup pencircle scaled stem_width;
  draw (0,h/2)--(w,h/2);
  draw (w/2,0)--(w/2,h);
endpic;
```

Finally, each font program should end with the `bye` command.

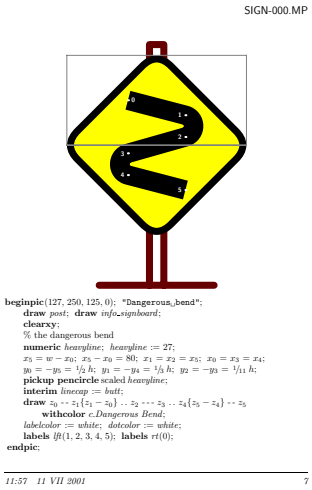


Fig. 4. Hardcopy proof of Signpost-500

```
bye
```

The last two steps are easy. We compile this file under LINUX with a command (or something analogous for other operating systems)†:

```
mpost -mem=type3 plus-000.mp
```

This step produces the font metric file and the PostScript commands which draw the plus shape. Finally, we collect the results of compilation into a Type 3 font with the PERL program:

```
mptot3 plus-000
```

To use ‘plus font’ in a TEX document it suffices to insert these lines:

```
\font\plusf=plus-000 at 10pt
\centerline{\plusf +\quad+ +++ +\quad+}
```

This code produces the seven pluses below.

+ +++++ +

A font cannot be proved faultless. If some glyphs are defective, the best way to correct them is to look at big hardcopy proof that shows what went wrong. For example, the hardcopy proof above could be be generated with the following shell commands:

```
mpost -mem=type3 '\mode=proof ; \
   input sign-000.mp'
tex \\input mproof sign-000.127
dvips mproof -o
```

† Unfortunately, the author does not know such commands for other operating systems.

Actually, the proof above contains some code which was pretty printed with `mft` tool (which is also a part of any TEX distribution).

MANAGING FONTS

Note that it is not wise to make one-time-only variation of a font by changing the font source. This kind of data multiplication resembles viruses spreading. To change font parameters `mode_setup` in conjuction with `change_mode` macro should be used. Again, I think that this concept is best explained by an example.

Assume that fictitious document `doc.tex` uses two fictitious Type 3 fonts named: *SuperFoo*, *SmashingBar*, and the font programs reside in the files `foo.mp`, `bar.mp`.

To re-parameterize these fonts create file `doc.mp` with the following content:

```
mode_def doc_foo =
  final_; % create metric file and execute:
  metapost code for SuperFoo
enddef;
mode_def doc_bar =
  final_;
  metapost code SmashingBar
enddef;
```

Then create font metric files, Type 3 fonts, and `dvips` fontmap files with the following commands (see Appendix B for an explanation):

```
mpost -mem=type3 \
   '\change_mode("doc","doc_foo"); \
   input foo.mp'
mptot3 -fontmap=foo.map foo.mp
mpost -mem=type3
   '\change_mode("doc","doc_bar"); \
   input bar.mp'
mptot3 -fontmap=bar.map bar.mp
```

It is convenient to concatenate fontmap files:

```
cat foo.map bar.map > doc.map
```

Now, we can compile `doc.tex` with:

```
tex doc.tex
```

and convert produced `doc.dvi` to PostScript with the command:

```
dvips -u doc.map doc.dvi -o
```

This should generate file named `doc.ps` which could be viewed and printed, for example with the `gv` program.

FONT DESIGN TO THE PEOPLE

Although the problems of letterform design are extremely subtle, more than most people think, because our machines and our eyes interact with the shapes in

complicated ways [8], these arguments do not necessarily apply to Type 3 fonts. Special purpose designs, for example geometric ones, could be programmed even by a rank one amateur designer and programmer. The article proves this last statement, I hope.

The font design is a fun. So, I decided to make available over a WEB all the tools and fonts created during preparation of this manuscript.

The master sources, which hopefully reached the beta stage, could be picked up from the following URL:

`ftp.gust.org.pl/pub/TeX/fonts/mtype3`.

There you will find the file named `README`. It contains detailed installation instructions. Then turn to the files called `Makefile`. Most of them are very simple. They encapsulate actions needed to create hardcopy proofs, fonts, etc. There are separate directories with examples of use and a booklet presenting all fonts.

REFERENCES

[1] Adobe Systems Incorporated. 1985. *Tutorial and Cookbook*. Addison Wesley, 221–228.

[2] Adobe Systems Incorporated. 1992. *The PostScript Font Handbook*. Addison Wesley.

[3] Adobe Systems Incorporated. 1999 (3rd ed.). *PostScript Language Reference Manual*. Addison Wesley.

[4] Per Cederqvist et al. 1993. *Version Management with CVS* (for version 1.10.8). Available online with the CVS package. Signum Support AB.

[5] Jeremy Gibbons. 1999. "Hey — it works!" (Hints & Tricks). *TUGboat* **20**, 367–370.

[6] John D. Hobby. 1992. *A user's Manual for MetaPost*. Technical Report 162. AT&T Bell Laboratories, Murray Hill / New Jersey. Available online as a part of METAPOST distribution.

[7] Bogusław Jackowski at all. 1999. "Antykwa Półtawskiego: a parametrized outline font". EuroTEX 99 Proceedings. Ruprecht-Karls-Univerität Heidelberg, 117–141.

[8] Donald E. Knuth. 1982. "The Concept of a Meta-Font". *Visible Language* **16**, 3–27.

[9] Donald E. Knuth. 1985. "Lessons Learned from METAFONT". *Visible Language* **19**, 35–53.

[10] Donald E. Knuth. 1986. *The METAFONTbook*. American Mathematical Society and Addison Wesley.

[11] Donald E. Knuth. 1988. "A Punk Meta-Font". *TUGboat* **9**, 152–168.

[12] Donald E. Knuth. 1988. "Virtual Fonts: More Fun for Grand Wizards". *TUGboat* **11**, 13–23.

[13] Donald E. Knuth. 1992. *Computer Modern Typefaces*. Addison Wesley.

[14] Donald E. Knuth. 1994. *The TEXbook*. American Mathematical Society and Addison Wesley.

[15] Richard M. Stallman and Roland McGrath. 2000. *GNU Make* (for version 3.79). Available online as a part of GNU MAKE package.
[16] Apostolos Syropoulos. 2000. The MF2PT3 tool. Available online from `www.obelix.ee.duth.gr/~apostolo`.
[17] La Rochefoucauld. 1655-1678. *Maxims*.
[18] George B. Shaw. 1903. *From the Revolutionist's Handbook*.
[19] Democritus. ca 400 B.C. *Ethical Precepts*.

APPENDIX A

Here the format of Type 3 font is described. This description is somehow simplified with the respect to examples to be found in [1] and [3].

Each font should begin with two lines of comments.

```
%!PS-Adobe-2.0: Square 1.00
%%CreationDate: 1 May 2001
```

A Type 3 font consists of a single dictionary, possibly containing other dictionaries, with certain required entries. The dictionary of size 99 should suffice for fonts which consists of several characters.

```
99 dict begin
```

This dictionary should include following entries:

- Variable `FontType` indicates how the character information is organized; for Type 3 fonts it has to be set 3.
- Variable `LanguageLevel` set to minimum language level required for correct behavior of the font.
- Array `FontMatrix` transforms the character coordinate system into the user coordinate system. This matrix maps font characters to one-unit coordinate space, which enables PostScript interpreter to scale font characters properly. This font uses 1000-unit grid.
- Array of four numbers `FontBBox` gives lower-left (l_x, l_y) and upper-right (u_x, u_y) coordinates of the smallest rectangle enclosing the shape that would result if all characters of the font were placed with their origins coincident, and then painted. This information is used in making decisions about character caching and clipping. If all four values are zero, no assumptions about character bounding box are made.

```
/FontType 3 def
/LanguageLevel 2 def
/FontMatrix [ 0.001 0 0 0.001 0 0 ] def
/FontBBox [ 0 0 1000 1000] def
```

`FontInfo` dictionary is optional. All info stored there is entirely for the benefit of PostScript language programs using the font, or for documentation.

- `FamilyName` — a human readable name for a group of fonts. All fonts that are members of such a group should have exactly the same FamilyName.

- `FullName` — unique, human readable name for an individual font. Should be the same name as one used when registering the font with definefont operator below.
- `Notice` — copyright, if applicable.
- `Weight` — name for the "boldness" attribute of a font.
- `version` — version number of the font program.
- `ItalicAngle` — angle in degrees counterclockwise from the vertical of the dominant vertical strokes of the font.
- `isFixedPitch` — if true, indicates that the font is a monospaced font; otherwise set false.
- `UnderlinePosition` — recommended distance from the baseline for positioning underlining strokes (y coordinate).
- `UnderlineThickness` — recommended stroke width for underlining, in units of the character coordinate system.

```
/FontInfo <<
   /FamilyName (Geometric)
   /FullName (Square)
   /Notice (Type 3 Repository.
     Copyright \(C\) 2001 Anonymous.
     All Rights Reserved.)
   /Weight (Medium)
   /version (1.0)
   /ItalicAngle 0
   /isFixedPitch true
   /UnderlinePosition 0.0
   /UnderlineThickness 1.0
>> def
```

Array `Encoding` maps character codes (integers) to character names. All unused positions in encoding vector must be filled with the name `.notdef`. It is special in only one regard: if some encoding maps to a character name that does not exist in the font, `.notdef` is substituted. The effect produced by executing `.notdef` character is at the discretion of the font designer, but most often it is the same as space.

```
/Encoding 256 array def
0 1 255 {Encoding exch /.notdef put} for
```

`CharacterProcedures` dictionary contains individual character definitions. This name is not special. Any name could be used, but this name is assumed by the `BuildGlyph` procedure below.

```
/CharacterProcedures 256 dict def
```

Each character must invoke `setcachedevice` or `setcharwidth` operator before executing graphics operators to define and paint the character. The `setcachedevice` operator stores the bitmapped image of the character in the font cache. However, caching will not work if color or gray is used. In such cases

the `setcharwidth` operator should be used. It is similiar to `setcachedevice` (explained below), but it declares that the character being defined is not to be placed in the font cache.

w_x w_y l_x l_y u_x u_y `setcachedevice` –
w_x, w_y — comprise the basic width vector, ie. the normal position of the origin of the next character relative to origin of this one
l_x, l_y, u_x, u_y — are the coordinates of this character bounding box

w_x w_y `setcharwidth` –
w_x w_y — comprise the basic width vector of this character

```
CharacterProcedures /.notdef {
   1000 0 0 0 1000 1000 setcachedevice
   1000 0 moveto
} put
Encoding 32 /space put
CharacterProcedures /space {
   1000 0 0 0 1000 1000 setcachedevice
   1000 0 moveto
} put
Encoding 83 /square put % ASCII 'S'
CharacterProcedures /square {
   1000 0 setcharwidth
   0 1 1 0 setcmykcolor % red
   0 0 1000 1000 rectfill
} put
```

Procedure `BuildGlyph` is called within the confines of a `gsave` and a `grestore`, so any changes `BuildGlyph` makes to the graphics state do not persist after it finishes.

`BuildGlyph` should describe the character in terms of absolute coordinates in the character coordinate system, placing the character origin at $(0, 0)$ in this space.

The Current Transformation Matrix (CTM) and the graphics state is inherited from the environment. To ensure predictable results despite font caching, `BuildGlyph` must initialize any graphics state parameters on which it depends. In particular, if `BuildGlyph` executes the `stroke` operator, it should explicitly set: dash parameters, line cap, line join, line width. These initializations are unnecessary, when characters are not cached, for example if the `setcachedevice` operator is not used.

When a PostScript language interpreter tries to show a character from a font, and the character is not already present in the font cache it pushes on the operand stack: *current font dictionary* and *character name*. The `BuildGlyph` procedure must remove these two objects from the operand stack and use this information to render the requested character. This typically involves finding the character procedure and executing it.

```
/BuildGlyph { % stack: font charname
  exch
  begin
  % initialize graphics state parameters
  % turn dashing off: solid lines
    [ ] 0 setdash
  % projecting square cap
    2 setlinecap
  % miter join
    0 setlinejoin
  % thickness of lines rendered by
  % execution of the stroke operator
    50 setlinewidth
  % the miter limit controls the stroke
  % operator's treatment of corners;
  % this is the default value and it
  % causes cuts off mitters at
  % angles less than 11 degrees
    10 setmiterlimit
    CharacterProcedures exch get exec
  end
} bind def
currentdict
end % of font dictionary
```

Finally, we register the font name as a font dictionary defined above and associate it with the key `Square`. Additionally the `definefont` operator checks if the font dictionary is a well-formed.

```
/Square exch definefont pop
```

If the following lines are not commented out the Ghostscript program (a public domain PostScript interpreter) will show the text below online. Obviously, these lines should be commented out in the final version of the font program.

```
/Square findfont
  72 scalefont setfont
0 72 moveto (S) show
showpage
```

METAPOST MACROS FOR TYPE 3 FONTS

TYPE 3 DRIVER FILE

This driver file serves as chief executive for the files which supports Type 3 font generation process.

When the equality below is true, this file has been input.

if *base_name* = `"type3"`: **endinput fi**

The first few lines usually give the base file a name and version number.

string *base_name*, *base_version*; *base_name* = `"type3"`; *base_version* = `"1.27"`;
message `"Preloading␣the␣type3␣mem␣file,␣version␣"` & *base_version*;

Supporting macros are divided into several files.

input `type3adm` % glyph and font administration.
input `type3mop` % modes of operation
input `type3pf` % support for hardcopy proofs
input `type3ps` % PostScript specific items.

endinput

GLYPH AND FONT ADMINISTRATION

Each glyph is build between **beginpic** . . . **endpic**. The **beginpic** was designed after plain **beginchar** macro. Each **beginpic** begins a group, which end at the next **endpic**. Then the given glyph code is stored and character box dimensions in `mpost` internal variables *charwd*, *charht*, *chardp* and it sets box dimensions w, h, and d. Finally it clears the z variables, the current picture, and the current pen. TEX needs to know the size of each characters's "bounding box". A total of four dimensions is given for each character:

- *charwd*, the width of the bounding box
- *charht*, the height above baseline of the bounding box
- *chardp*, the depth below baseline of the bounding box
- *charic*, the character "italis's correction".

The `mpost` records the value of its internal quantities, and writes them onto `tfm` file, at the time of **shipit** command.

def # = /(*grid_size*/*designsize*) **enddef**;

def beginpic(**expr** *c*, *width*, *height*, *depth*) = % character code *c*
  begingroup
  charcode := **if** known *c*: byte *c* **else**: 0 **fi**;
  w := *width*; *h* := *height*; *d* := *depth*;
  charic := 0; **clearxy**; **clearit**; **clearpen**;
  drawoptions();
  scantokens *extra_beginpic*;
enddef;

def italcorr expr *x* = **if** *x* > 0: *charic* := *x*# **fi enddef**;

newinternal *proofscale*; *proofscale* := 1;

Glyph widths are written onto file named ⟨jobname⟩.pcw. These widths are read by `mptot3` script which uses them as parameters to the PostScript `setcharwidth` operator.

```
def endpic = scantokens extra_endpic;
    write decimal charcode & ":" & decimal w to jobname & ".pcw";
    charwd := w#;  charht := h#;  chardp := d#;
    if proofing > 0: makebox(proofrule);
        currentpicture := currentpicture scaled proofscale;
    fi
    shipit;
    endgroup
enddef;

def shipit = if proofing ≥ 0:
    shipout currentpicture transformed (identity shifted (xoffset, yoffset)) fi
enddef;
newinternal xoffset, yoffset;

string extra_beginpic, extra_endpic;
extra_beginpic = extra_endpic = "";
```

The *designsize* of a font is another internal quantity that is output to `tfm` file. When a TEX user asks for a font '`at`' a certain size, the font is scaled by the ratio between the "at size" and the "design size".
The *designsize* must be at least 1 pt and must be less than 2048 pt and every dimension of the font should be less than 16 times the design size in absolute value.
The "design size" is an imprecise notion, because there need be no connection between the design size and any specific measurement in a font. It is something like dress size or shoe sizes. **For Type 3 fonts we set the design size to 100 bp**, which seems to be a good compromise between the accuracy of the `mpost` calculations and the maximum size of a grid.

```
designsize := 100;
```

It is suggested that fonts use a 1000-unit grid. This is the default grid size used in Type 1 fonts programs.

```
newinternal grid_size;  grid_size := 1000;
```

The other type information that appears in `tfm` file applies to a font as a whole. These include numeric data specified in "fontdimen" commands. Note that math symbols fonts are required to have at least 22 fontdimen parameters and math extensions at least 13.

```
def font_slant expr x = fontdimen 1: x enddef; % no hash here!
def font_normal_space expr x = fontdimen 2: x# enddef;
def font_normal_stretch expr x = fontdimen 3: x# enddef;
```

```
def font_normal_shrink expr x = fontdimen 4: x# enddef;
def font_x_height expr x = fontdimen 5: x# enddef;
def font_quad expr x = fontdimen 6: x# enddef;
def font_extra_space expr x = fontdimen 7: x# enddef;

def font_identifier expr x = font_identifier_ := x enddef;
def font_coding_scheme expr x = font_coding_scheme_ := x enddef;

string font_identifier_, font_coding_scheme_;
font_identifier_ = font_coding_scheme_ = "UNSPECIFIED";
```

TEX relies on lots of parameters when it typesets math formulas. He will not typeset a math formula unless symbol fonts contain at least 22 fontdimen parameters.

```
vardef font_num @# expr x = if (@# < 1) or (@# > 3):
        errmessage "Wrong␣suffix␣to␣font_num:␣" & decimal @#
    else: fontdimen 7 + @#: x# fi
enddef;

vardef font_denom @# expr x = if (@# < 1) or (@# > 2):
        errmessage "Wrong␣suffix␣to␣font_denom" & decimal @#
    else: fontdimen 10 + @#: x# fi
enddef;

vardef font_sup @# expr x = if (@# < 1) or (@# > 3):
        errmessage "Wrong␣suffix␣to␣font_sup" & decimal @#
    else: fontdimen 12 + @#: x# fi
enddef;

vardef font_sub @# expr x = if (@# < 1) or (@# > 2):
        errmessage "Wrong␣suffix␣to␣font_sub" & decimal @#
    else: fontdimen 15 + @#: x# fi
enddef;

def font_sup_drop expr x = fontdimen 18: x# enddef;
def font_sub_drop expr x = fontdimen 19: x# enddef;

vardef font_delim @# expr x = if (@# < 1) or (@# > 2):
        errmessage "Wrong␣suffix␣to␣font_delim" & decimal @#
    else: fontdimen 17 + @#: x# fi
enddef;

def font_axis_height expr x = fontdimen 22: x# enddef;
```

Extension fonts should contain at least 13 fontdimen parameters.

```
def font_default_rule_thickness expr x = fontdimen 8: x# enddef;

vardef font_big_op_spacing @# expr x = if (@# < 1) or (@# > 5):
        errmessage "Wrong␣suffix␣to␣font_big_op_spacing"&decimal @#
```

```
    else: fontdimen 8 + @#: x# fi
enddef;

endinput
```

MODES OF OPERATION

The standard way to create a Type 3 font is to start up the `mpost` program by saying

`mpost -mem=type3 \mode=`⟨mode name⟩`; input` ⟨font program⟩

and afterwards to collect glyphs created by `mpost` into a Type 3 font with Perl script

`mptot3` ⟨font program⟩

The `mode` is omitted if `mode=final`. The mode name should have been predeclared in your base file, by the **mode_def** routine below. If, however, you need special modes that aren't in the base, you can put its commands into a file (e.g., '`specmodes.mp`') and invoke it by saying

`mpost -mem=type3 \change_mode("specmodes",`⟨mode name⟩`);`
`input` ⟨font program⟩

instead of giving a predeclared mode name.

Here is the **mode_setup** routine, which is usually one of the first macros to be called after a font program establishes the values of all font parameters. The first '**scantokens**' in **mode_setup** either reads a special file or calls a macro that expands into commands defining the mode.

```
transform currenttransform; def t_ = transformed currenttransform enddef;

def mode_setup = if unknown mode: mode = final; fi
    numeric aspect_ratio; transform currenttransform;
    if unknown aspect_ratio: aspect_ratio = 1; fi
    if string mode: scantokens("input␣" & mode);
        for i := 1 upto number_of_modes:
            if mode_name[i] = requested_mode_: scantokens mode_name[i]; fi
        endfor
    else: scantokens mode_name[mode];
    fi
    scantokens extra_setup; % the user's special last-minute adjustments
    currenttransform :=
        if unknown currenttransform: identity else: currenttransform fi
            yscaled aspect_ratio;
    clearit;
enddef;

def change_mode(expr file_name, mode_name) =
    string mode; mode := file_name;
```

```
    requested_mode_ := mode_name & "_"
enddef;
string requested_mode_;

string extra_setup, mode_name[];
extra_setup = ""; % usually there's nothing special to do
```

When a mode is defined (e.g., '`proof`'), a numeric variable of that name is created and assigned a unique number (e.g., 1). Then an underscore character is appended, and a macro is defined for the resulting name (e.g., '`proof_`'). The *mode_name* array is used to convert between number and name (e.g., $mode_name_1$ = `proof_`).

```
def mode_def suffix $ =
    $ := incr number_of_modes;
    mode_name[$] := str $ & "_";
    expandafter quote def scantokens mode_name[$]
enddef;
newinternal number_of_modes;
```

Three basic modes are now defined, starting with two for proofing.

Proof mode — for initial design of characters.

```
mode_def proof =
    proofing := 2; % yes, we're making full proofs
    fontmaking := 0; % no, we're not making a font
    tracingtitles := 1; % yes, show titles online
enddef;
```

Smoke mode — for label-free proofs.

```
mode_def smoke =
    proof_; % same as proof mode, except:
    proofing := 1; % yes, we're making unlabeled proofs
    fontmaking := 0; % no, we're not making a font
    let makebox = maketicks; % make the boxes less obtrusive
enddef;
```

Final mode — a typical mode for font generation (note, that we get a TEX font metric file if `mpost` internal quantity *fontmaking* is positive at the end of our job).

```
mode_def final =
    proofing := 0; % no, we're not making proofs
    fontmaking := 1; % yes, we are making a font
    tracingtitles := 0; % no, don't show titles at all
    prologues := -2; % high resolution bounding box.
enddef;
```

```
newinternal grayproofing;

mode_def grayproof =
    proofing := 2; % yes, we're making full proofs
    fontmaking := 0; % no, we're not making a font
    tracingtitles := 1; % yes, show titles online
    grayproofing := 1; % use 'proofcolor' for drawing
enddef;

localfont := final; % the mode most commonly used to make fonts
```

It is not likely that additional modes are needed, but if they are, additional mode_def commands should be in another input file that gets loaded after the plain base. The auxiliary file should set *base_version* := *base_version* & `"/localname"`

PROOF LABELS AND RULES

The next main section of `type3.mp` is devoted to macros for the annotations on proofsheets.

```
newinternal proofing; % < 0 to supress output; > 1 to do labels
color proofcolor; % color for output when proofing > 0
proofcolor = .3[white, black];
color background; background = white;
color dotcolor, labelcolor; dotcolor = black; labelcolor = black;

newinternal defaultdotsize; defaultdotsize := 3;
newinternal defaultrulethickness; defaultrulethickness := 1;
```

Labels are generated at the lowest level by **makelabel** and **makepiclabel**:

Put string *s* near point *z*.

```
vardef makelabel @#(expr s, z) = picture p;
    if proofing > 1: if known z:
            p = s infont defaultfont scaled defaultscale;
            draw p shifted
  (z + labeloffset * laboff_@# -
  (labxf_@# * lrcorner p + labyf_@# * ulcorner p + (1 - labxf_@# - labyf_@#) * llcorner p))
            withcolor labelcolor;
            interim linecap := rounded;
            draw z withpen pencircle scaled defaultdotsize
              withcolor dotcolor;
        fi
    fi
enddef;
```

Put string *s* near point *z* shifted by *shift* and scaled by *scale*.

```
vardef makepiclabel @#(expr s, z, shift, scale) =
```

```
    save zz; pair zz; zz = z shifted shift scaled scale;
    makelabel @#(s, zz);
enddef;
```

Users generally don't invoke **makelabel** directly, because there's a convenient shorthand. For example, '**labels**(1, 2, 3)' expands into '**makelabel**("1", z_1); **makelabel**("2", z_2); '**makelabel**("3", z_3)' (But nothing happens if *proofing* $\le$ 1.)

```
vardef labels @#(text t) =
    forsuffixes $ = t: makelabel @#(str $, z$); endfor
enddef;

vardef piclabels @#(expr shift, scale)(text t) =
    forsuffixes $ = t: makepiclabel @#(str $, z$, shift, scale); endfor
enddef;

vardef penlabels @#(text t) = forsuffixes $$ = l, , r: forsuffixes $ = t:
            makelabel @#(str $$$, z$.$$); endfor endfor
enddef;

vardef picpenlabels @#(expr shift, scale)(text t) =
    forsuffixes $$ = l, , r: forsuffixes $ = t:
            makepiclabel @#(str $$$, z$.$$, shift, scale); endfor endfor
enddef;
```

When there are lots of purely numeric labels, you can say, e.g.,

labels(1, **range** 5 **thru** 9, **range** 100 **thru** 124, 223)

which is equivalent to '**labels**(1, 5, 6, 7, 8, 9, 100, 101, . . . , 124, 223)'. Labels are omitted from the proofsheets if the corresponding z value isn't known, so it doesn't hurt (much) to include unused subscript numbers in a range.

```
def range expr x = numtok[x] enddef;
def numtok suffix x = x enddef;
tertiarydef m thru n = m for x = m + 1 step 1 until n: , numtok[x] endfor
enddef;
```

A straight line will be drawn on the proofsheet by **proofrule**.

```
def proofrule(expr w, z) =
    begingroup interim linecap := squared;
    draw w .. z withpen pencircle scaled defaultrulethickness
    withcolor proofcolor
    endgroup
enddef;
```

You can produce lots of proof rules with **makegrid**, which connects an arbitrary list of x coordinates with an arbitrary list of y coordinates:

```
def makegrid(text xlist, ylist) =
    xmin_ := min(xlist); xmax_ := max(xlist);
    ymin_ := min(ylist); ymax_ := max(ylist);
    for x = xlist: proofrule((x, ymin_), (x, ymax_)); endfor
    for y = ylist: proofrule((xmin_, y), (xmax_, y)); endfor
enddef;

vardef labelfont suffix $ = defaultfont := str $ enddef;

def makebox(text r) =
    for y = 0, h, -d: r((0, y), (w, y)); endfor
    for x = 0, w: r((x, -d), (x, h)); endfor
enddef;

def maketicks(text r) =
    for y = 0, h, -d: r((0, y), (w/10, y)); r((w - w/10, y), (w, y)); endfor
    for x = 0, w: r((x, h/10 - d), (x, -d)); r((x, h - h/10), (x, h)); endfor
enddef;
```

MACROS FOR FILLING AND ERASING

```
def pc_ = hide(if grayproofing > 0: def pc_ = do_pc_ enddef; else: def pc_ =
enddef; fi) pc_
enddef;
def do_pc_ = withcolor proofcolor enddef;

def fill expr c = addto currentpicture contour c t_ _op_ pc_ enddef;

def draw expr p = addto currentpicture
    if picture p: also p
    else: doublepath p t_ withpen currentpen
    fi _op_ pc_
enddef;

def filldraw expr c = addto currentpicture contour c t_ withpen currentpen
    _op_ pc_
enddef;
def drawdot expr z = addto currentpicture contour makepath currentpen
    shiftedz t__op_ pc_
enddef;

endinput
```

TYPE 3 SPECIFIC ITEMS

One of required entries in each font dictionary is the array **FontMatrix**. This array is used by a PostScript interpreter to transform *glyph coordinate system*

into the user system coordinate system. This matrix maps font characters to one-unit coordinate space, which enables PostScript interpreter to scale font characters properly. It is suggested that fonts use a 1000-unit grid. This implies the following **FontMatrix**:

FontMatrix [.001 0 0 .001 0 0] **def**

But what matrix is used is up to user, as long as the **FontMatrix** would be adjusted accordingly. The **FontMatrix** will be read from a file named ⟨jobname⟩.par. The parameters written to this file are read by the `mptot3` script.

A two dimensional transformation is described in math as 2×3 matrix

$$\begin{bmatrix} a & b & t_x \\ c & d & t_x \end{bmatrix}$$

In the PostScript language, this matrix is represented as a six-element array object

$$[\ a\ \ b\ \ c\ \ d\ \ t_x\ t_y\ ]$$

For example, scaling by the factor s_x in the x dimension and s_y in the y dimension is accomplished by the matrix:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \end{bmatrix}$$

or by an array object:

$$[\ s_x\ \ 0\ \ 0\ \ s_y\ \ 0\ 0\ ]$$

def font_matrix(**expr** a, b, c, d, tx, ty) = **write** "FontMatrix␣[␣"&
decimal a & "␣" & decimal b & "␣" & decimal c & "␣" & decimal d&
"␣" & decimal tx & "␣" & decimal ty & "␣]" **to jobname** & ".par"
enddef;

Each PostScript font has a name and belongs to some family, has attached version etc. These parameters are written onto *jobname.par* file too.

def font_name expr *name* = **write** "FontName␣"&*name***tojobname**&".par"
enddef;
def family_name expr *name* = **write** "FamilyName␣" & *name* **to jobname** &
".par"
enddef;
def font_version expr x = **write** "FontVersion␣" & x **to jobname** & ".par"
enddef;
def is_fixed_pitch expr b = **write** "isFixedPitch␣"&
if b: "true" **else**: "false" **fi to jobname** & ".par"
enddef;

endinput

♢ ♢ ♢

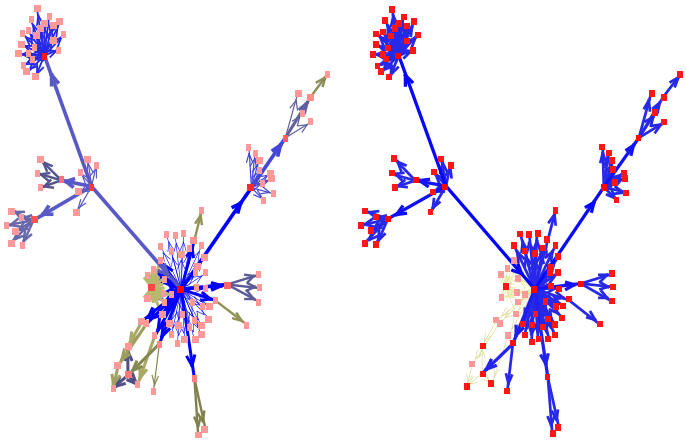
Literate Programming not just another pretty face

M.A. Guravage *(NLR)*

ABSTRACT. The structure of a software program may be thought of as a "web" that is made of many interconnected pieces. To document such a program, we want to explain each individual part of the web and how it relates to its neighbors. – D.E.K.

My association with literate programming was love at first sight. I admired the crisp form and clear content of the programs I read. I simply became jealous; I wanted my programs to look and work like that. When I first read Professor Knuth's description of a program as a "web" of interconnections, I became curious whether the pattern of connections particular to literate programs could be modeled, analyzed, and quantified.

The most obvious model to represent information about the relationship between the pieces of a web is a graph. Graphs have been used extensively to model all sorts of relationships, and the theory of graph metrics is well developed. The trick is to interpret the various standard graph metrics, e.g. flow, width, depth, size, and edge-to-node ratio, in the context of a web's literate structure. If the same can be done for a traditional program's call-graph structure, it might be possible to compare objectively literate and traditional programming styles.

The two graphs below represent two different metrics applied to the same literate program. The left graph was created using an 'activation' metric; that correctly identified, in thick blue, the main conceptual threads through the program. The right graph was created using an 'upward flow' metric; that colored yellow the paths to nodes most often reused.

Using graph visualization software developed at CWI in Amsterdam, I hope to show how various metrics can shed light onto the structure of both literate and traditional programs. For example, a measure called 'tree impurity' can tell us how far a graph deviates from being a tree, might allow us to compare literate and traditional programming styles.